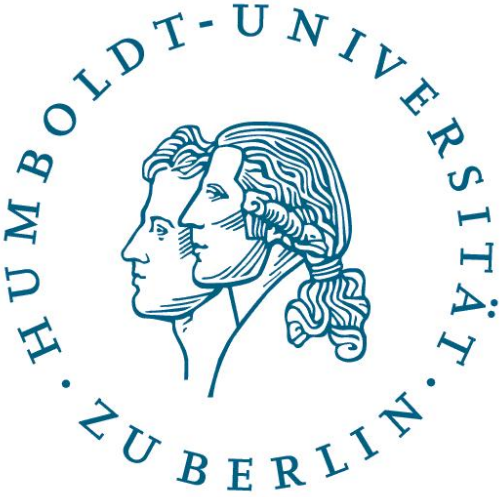
Humboldt-Universitaet zu Berlin
Wirtschaftswissenschaftliche Fakultaet



Simulation von Rohdaten

Simulation of raw data

Masterarbeit

Zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

in MEMS

an der Wirtschaftswissenschaftlichen Fakultaet

der Humboldt-Universitaet zu Berlin

vorgelegt von

Olzhas Kozbagarov

Matrikel-Nr 522155

Pruefer: Prof. Dr. Wolfgang Haerdle

Berlin, 23.10.2013

Contents

1 Introduction

The aim of work is the following: we are given a data set and we want to reproduce a new data set on a base of old one and this new data set should preserve some properties of given data set; namely, new data set should conserve some relationships from old data set such that if we perform factor analyses on two sets we should get close values of factor loadings. But records from new data set must be distinct from records of old one; that is, there are no identically same records in two sets. The original data set is students evaluations of lecturers; every record consists of 42 variables, each variable is binary or categorical or discrete (not necessary integer), but there can be missing values. (Due to reasons of confidentially, I worked with not entirely original data set, but with modified data set where all missing values were already imputed.) My data set consists of 2972 records.

To simulate new data set, I created application in Matlab using special software implemented in Matlab called "guide". Using this application, a user can simulate data by pushing corresponding buttons and there is no need for him or her to know names of all functions, take a track of all parameters. The interface of my application is shown on the following picture.

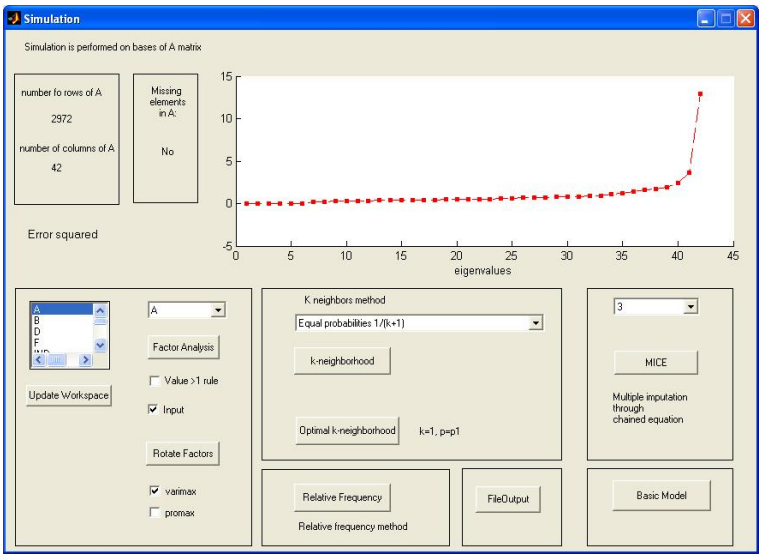


Figure 1: Interface of "Simulation" application

To start use of the application, we just write line "guide" in command window of Matlab and afterwards window appears. There we have to choose directory where the function *Simulation.m*, which contains code is stored. The original data set has to be in workspace of Matlab before any execution and the name for a data set must be assigned as A matrix, if it is not the case error

message appears. The code behind the application is written in function called *Simulation.m*. The following code is executed when we activate the application. The opening function does all necessary starting initialization: determines dimension of data set A, sets all check boxes, list boxes on default values, updates pop-up menus, and, in addition, determines if there are missing elements in matrix A. If we have missing values, we use expectation-maximization algorithm for maximum likelihood estimation of covariance matrix of A. The function *PlotEigErr* is called in the body of the program to find all eigenvalues of data set A (*lam* variable) and to plot them.

```
% --- Executes just before Simulation is made visible.
function Simulation_OpeningFcn(hObject, eventdata, handles,
    varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to Simulation (see VARARGIN)

% find matrix A in Matlab's workspace (matrix A contains data
% set); if there is no such matrix in workspace then error message
% appears
try
    A=evalin('base','A');
catch
    error;
end

% data set is stored under new global variable
% handles.current_data
handles.current_data=A;

% dimension of matrix A
handles.m=size(A,1);
handles.n=size(A,2);

% information about number of rows and number of columns
% of A appears on the applications form
set(handles.mytext1,'String',handles.m)
set(handles.mytext2,'String',handles.n)

% set initial values for check boxes
set(handles.checkbox2,'Value',1)
set(handles.checkbox3,'Value',1)

% update information for list box and pop-up menu
update_listbox(handles);
update_popupmenu(handles);

% determine if we have missing values; dmissing is boolean
% variable that is true when data set has at least one missing
% value
dmissing=false;
h=0;
for i=1:handles.m
```

```
    for j=1:handles.n
        if isnan(A(i,j))
            dmissing=true;
            h=h+1;
        end%{if}
    end%{for j}
end%{for i}

% global variable to show if we have missing values
handles.dmissing=dmissing;
if dmissing
    set(handles.mytext21,'String',h);
else
    set(handles.mytext21,'String','No');
end%{if}

for i=1:handles.n
    lamda(i)=1;
end%{for i}
% find correlation matrix of data set A and it's eigenvalues ;
% if matrix has missing values, we use maximum likelihood
% EM method to find covariance matrix;
[lam,error]=PlotEigErr(A,lamda,handles.dmissing,handles.n,'red','--
    rs')

% handles.lamda global variable for all eigenvalues of correlation
% matrix of data set A
handles.lamda=lam;

% Choose default command line output for Simulation
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);
```

One of methods to perform factor analysis, that is based on eigenvalues and eigenvectors of covariance matrix and is called principal component factor analysis, was used to find factor loadings. The eigenvalues of the correlation matrix that have value greater than one (9 eigenvalues) and factor loadings of 9 factors of data set A found by PC method are shown in the following tables. Large values of factors are highlighted by different colors. The first factor (blue) is responsible for 15 variables, the second (red) is for 5 variables. The rest variables are combinations of some two factors (green and yellow). The 9 eigenvalues account for proportion $\frac{\lambda_1+\cdots+\lambda_9}{42} = 67\%$ of the total sample variance.

12.92	3.59	2.46	1.93	1.75	1.61	1.41	1.19	1.06

Table 1: The first 9 largest eigenvalues of correlation matrix of original data set A

$\mathbf{F_1}$	$\mathbf{F_2}$	$\mathbf{F_3}$	$\mathbf{F_4}$	$\mathbf{F_5}$	$\mathbf{F_6}$	$\mathbf{F_7}$	$\mathbf{F_8}$	$\mathbf{F_9}$
-0,09	-0,12	-0,04	0,14	0,07	0,21	0,16	-0,53	0,08
-0,02	-0,10	0,13	-0,08	-0,04	0,05	-0,16	0,12	-0,80
-0,01	0,23	0,11	0,01	-0,16	0,08	-0,14	0,37	0,44
0,74	0,04	0,09	0,08	0,01	-0,17	-0,06	-0,07	0,12
0,13	-0,14	0,14	0,33	-0,02	0,50	-0,58	-0,06	-0,01
0,02	-0,10	0,08	0,26	-0,06	0,45	-0,69	-0,03	0,07
0,78	0,05	0,06	0,09	0,04	-0,28	-0,15	-0,03	0,08
0,77	0,11	-0,04	0,10	-0,01	-0,25	-0,11	-0,08	0,04
0,63	-0,09	-0,19	0,12	-0,14	-0,26	-0,14	-0,09	-0,06
0,74	-0,02	0,03	0,00	0,03	-0,24	-0,18	-0,09	0,04
0,63	-0,10	0,15	0,06	0,15	-0,04	0,01	0,00	-0,01
0,61	-0,08	-0,07	-0,20	0,45	-0,01	-0,06	0,11	0,06
0,68	-0,08	-0,05	-0,12	0,32	-0,14	-0,12	0,07	0,07
0,45	-0,06	-0,25	-0,19	0,42	0,13	-0,12	0,14	-0,01
0,70	-0,05	-0,05	0,13	0,01	-0,09	-0,01	-0,23	0,01
0,70	-0,02	-0,11	0,12	-0,05	-0,15	-0,05	-0,25	0,00
0,63	-0,09	-0,02	-0,05	0,05	-0,03	-0,09	-0,09	-0,19
0,64	0,05	-0,08	0,01	-0,01	-0,01	0,02	-0,05	-0,03
0,62	-0,10	-0,38	0,17	-0,26	0,09	0,09	0,00	-0,04
0,48	-0,10	-0,55	0,11	-0,32	0,20	0,13	0,20	-0,04
0,49	-0,16	-0,53	0,07	-0,23	0,16	0,09	0,17	0,05
0,63	-0,15	-0,14	-0,04	0,08	0,11	0,11	0,03	-0,09
0,41	0,00	-0,21	-0,21	0,37	0,26	0,12	0,03	-0,07
-0,09	-0,75	0,08	-0,08	-0,01	-0,09	0,02	0,03	0,13
-0,18	-0,70	0,01	-0,13	-0,12	-0,12	-0,09	0,08	0,04
-0,24	-0,80	0,04	-0,14	-0,03	-0,01	-0,01	0,01	0,08
0,54	-0,02	0,45	0,23	0,02	0,20	0,27	0,20	-0,08
0,57	-0,19	0,43	0,19	0,03	0,23	0,20	0,14	-0,03
0,71	-0,06	0,31	0,18	-0,03	0,13	0,19	0,06	0,05
-0,06	0,30	-0,06	-0,19	-0,01	-0,17	-0,21	0,42	0,16
-0,11	-0,69	0,08	-0,11	-0,02	-0,06	-0,02	-0,01	-0,02
0,44	0,30	0,11	-0,58	-0,31	0,13	-0,06	-0,08	-0,03
0,76	-0,04	0,39	0,03	-0,08	0,03	0,12	0,08	0,04
0,49	-0,02	0,16	-0,56	-0,31	0,15	-0,07	-0,14	0,03
0,77	0,02	0,31	-0,05	-0,08	0,03	0,05	0,03	0,07
-0,20	-0,95	0,07	-0,15	-0,06	-0,09	-0,03	0,03	0,07
0,55	0,17	0,15	-0,67	-0,37	0,16	-0,07	-0,13	0,00
0,91	-0,01	-0,07	0,12	-0,03	-0,27	-0,14	-0,17	0,02
0,67	-0,06	-0,25	-0,28	0,57	0,19	-0,03	0,12	-0,01
0,65	-0,14	-0,59	0,14	-0,33	0,18	0,13	0,15	-0,01
0,82	-0,07	0,46	0,14	-0,04	0,15	0,20	0,12	0,01
0,20	0,00	0,11	0,11	-0,21	-0,46	-0,19	0,38	-0,24

Table 2: The factor loadings for 9 factors of original data set A

We want to simulate a data set, which eigenvalues of correlation matrix have close values to the eigenvalues of matrix A. So I expect, when we perform factor analysis to this kind of data set, that factor loadings of original and simulated data sets are also close. To simulate new data set, I used four methods. The first method is k-neighborhood method where Gower distance is chosen as distance to determine neighbors. The second method uses conditional probabilities to simulate elements of certain variables where conditional probabilities are estimated by relative frequencies calculated using original data set. The factor model definition with assumptions on factor distribution, number of factors and factor loadings is employed in the third method. The fourth method is multiple imputation by chained equations. The intermediate steps in imputation use stochastic regression method with simulated normal errors and predictive mean matching principle to impute new elements.

The code behind all methods is shown after each method. I tried to write code in optimal way, every parts of code that is repeated more than one time, if it is possible, I wrote as separated function, also for reasons not to complicate code's reading, I wrote some parts of function's code as separate functions.

2 The first method: k-neighborhood estimation

The idea behind the first method is very simple. For every record in original data set, we find it's k nearest neighbors; after to simulate a new record, we assign positive probability weights to initial and it's k neighbor records and then using random independent draws number of variables times with given probabilities, we randomly choose one of $k+1$ values of records in certain variable to fill each variable of new record. The code written to implement this idea takes into account possibility of missing values, that is, if records on which bases simulation occurs have missing values than there is positive probability that new simulated record has missing values. After we get simulated record, we check if it coincides with records from the initial data set. If simulated record is distinct, we go to the next record; otherwise, we re-simulate until we get distinct record. We need some measure of distance to determine neighbors, we choose Gower distance. The choice of this distance is explained by possibility of application of the distance to data sets with missing elements. The formula to calculate Gower distance is the following:

$$\text{distance between } j \text{ and } h = \frac{1}{n}\sum_{i=1}^{n}\frac{(s_{ji} - s_{hi})}{\text{range of i column}}$$

If we calculate distance between two records using this formula and they have in a corresponding column at least one missing element, then this column i gives zero to total sum. Also, if all elements of any column of original matrix A are equal, this column gives zero to total sum. The range of that column in this case is zero and we should take this into account in code to avoid division

by zero. It could happen that there are identical records in matrix A or there are maybe records identical in all columns except some columns where there are missing elements. In those cases, the distance between these records is zero. We can say that such records compose class of records. So any class of records consists of records such that distance between any two records from the same class is zero. The function written to find record's k neighbors finds one record from each of k closest classes, so the distance between any two neighbors is always positive.

When we do simulations, we are free to choose the number of neighbors and probability weights; intuitively, it looks more reasonable to give more probability to initial and use not too many neighbors, but enough number of them to provide variability in data to get new distinct records.

The button called *k neighborhood* on the application form performs the simulation using k neighborhood method. A user can choose four different types of assigning probabilities from pop-up menu above the button: with equal probabilities where all records get equal weight $\frac{1}{k+1}$; decreasing probabilities $\frac{1}{s}, \frac{1}{2s}, \cdots, \frac{1}{(k+1)s}$ where more weight is assigned to initial record and closest neighbors, s is chosen such that sum of probabilities is one; increasing probabilities $\frac{1}{(k+1)s}, \frac{1}{ks}, ..., \frac{1}{s}$ where less weight is assigned to initial record and closest neighbors and decreasing distance dependent probabilities $\frac{(s-d1)}{s}, ..., \frac{(s-dk)}{s}$ where $d_1 < d_2 < \cdots < d_k$ and d_i is gower distance between initial record and it's i-th neighbor. When a user has chosen probability weights and pushes the button, dialog box appears where a user has to input the desirable number of neighbors. The result of execution is matrix S with simulated records and it is stored in Matlab's workspace by S name. In addition, square error (sum of squared differences of corresponding eigenvalues of correlation matrices of A and S) is calculated and all eigenvalues of correlation matrix of S are illustrated on the application form.

When an user pushes the button, the function *pushbutton callback* is executed. The code asks a user to input number of neighbors' value and verifies if integer positive number has been inputted; otherwise, error message appears. The function *InputNum* performs that. If everything is right, the function *GowerDistance* is called, which finds matrix with calculated distances. The result of execution of this function is symmetric matrix D, which size is m times m (where m is number of records of data set A, $m = 2972$), all diagonal elements are zeros, because they represent distance with itself. This function also finds $k+1$ times m matrix IND, each column contains indexes of k closest neighbors in matrix A. That is, for each record from 1 to m, it shows place of it's k neighbors in matrix A. The row dimension is $k+1$ because the first element just shows record's own index in matrix A.

Before pushing the button, a user has chosen desirable probability weights in the pop-up menu. The code, in dependence of selection in the pop-up menu, calls function *Simulation*1 or 2 or 5 or 6. Every function is different from each other only in way of defining probabilities. They all have in body of program common function *SimulationMainBlock*, which takes defined probabilities and

does all simulations. The simulations are based on modeling random variable, that takes discrete values from 1 to $k+1$ with given probabilities, from uniform random variable on interval $[0,1]$. (We assume that the Matlab's function rand generates random variable with true uniform distribution.)

There is possibility for occurrence of the following problem: we simulate a record based on some concrete record and after 10000 tries, we may not get distinct record (the choice of number 10000 was arbitrary and can be changed to different number in function *SimulationMainBlock* (if $g > 10000$)). For example, when number of neighbors is one, initial record and it's neighbor might be identical except one column. Then it is impossible to get record, which is different from the two given records under given principle of simulation. The solution of this obstacle is done by the following method: if we do not get new distinct record after 10000 iterations, we add to this concrete record one additional neighbor, that is, we consider instead of chosen k neighbors $k+1$ ones. We continue simulations with new number of neighbors for this given record and, if again after 10000 iterations, we can't get distinct record, we add the next neighbor and so on. The indexes of such problematic records are written during execution in workspace of Matlab ($j =$) and the number of times it appears denotes the number of added additional neighbors. The function *ReAssignProb* is called in the body of program *SimulationMainBlock* in case we meet this problem.

Every time we get simulated record, the function *Check*2, in the body of program *SimulationMainBlock*, verifies if it is not identical to some records in A and it returns boolean value true in case it is distinct. The function *SimulationMainBlock* terminates when we have n simulated records. Then we go to the last lines of function *pushbutton callback*, where the function *PlotEigErr* calculates eigenvalues of S and plots them with sum of squared errors. The code behind the button and all embedded functions with comments is shown below.

```
% --- Executes on button press in pushbutton7.
function pushbutton7_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton7 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% input the number of neighbors and check if it is positive integer
str='Enter the number of neighbors';
type='integer';
answer=InputNum(str,0,200,type);
if ~isempty(answer) & (answer~=-1)
                % starting initializations
                k=answer;
                A=handles.current_data;
                n=handles.n;
                m=handles.m;
                lamda=handles.lamda;
                dmissing=handles.dmissing;
                % we calculate distances and determine neighbors for
                % all records of A
                [D,IND,KN]=GowerDistance(A,m,n,200);
```

```
                % a user has chosen in pop-up menu desirable
                % probability weights
                popupVal=get(handles.popupmenu2,'Value')

                % the result of simulation is matrix S
                switch popupVal
                    case 1%'Equal probabilities'
                        S=Simulation1(A,IND,KN,m,n,k);
                    case 2%'Decreasing probabilities'
                        [S]=Simulation2(A,IND,KN,m,n,k);
                    case 3%'Increasing probabilities'
                        [S]=Simulation6(A,IND,KN,m,n,k);
                    case 4%'Distance dependent probabilities'
                        [S]=Simulation5(A,IND,KN,m,n,k);
                end%{switch}

                % check if S has missing elements
                if dmissing
                    dmissing=false;
                     for i=1:m
                            for j=1:n
                                 if isnan(S(i,j))
                                     dmissing=true;
                                  end%{if}
                            end%{for j}
                    end%{for i}
                end%{if dmissing}

                % result of simulation is matrix S
                assignin('base','S',S);
                % plot eigenvalues and squared error
                [lam,error]=PlotEigErr(S,lamda,dmissing,n,'black','--
                    bo');
                % calculated error is shown on application form
                set(handles.text16,'String',error);
                msgbox('Output is matrix S');
end%{if ~isempty(answer)}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [D,IND,KN]=GowerDistance(A,m,n,k)
%Input - A is m times n matrix containing data set (every row is
%            an record containing n different variables)
%        m is number of rows in A (number of records)
%        n is number of columns in A (number of variables in
%            every record)
%        k is number of neighbors
%Output - D is m times m matrix containing calculated Gower
%            distances for each record with other m-1 records
%        KN  is k+1 times m matrix containing distance values
%            of k least distant records
%           (the first row of KN contains distance with itself)
%        IND is k+1 times m matrix containing indexes of k
%            closest neighbors in A matrix; the first row
%            of IND contains record's own index
```

```
% we find the range of elements for every column of matrix A;
% i element of array rng shows index of column in matrix A
for i=1:n
    % we look for the first not missing element in column i of A
    b=true;
    h=1;
    while (h<n+1) & (b)
        if isnan(A(h,i))
            h=h+1;
        else
            maxA=A(h,i);
            minA=A(h,i);
            b=false;
        end%{if}
    end%{while}

    % determine max and min of every variable
    for j=1:m
        if ~isnan(A(j,i)) & (A(j,i)>maxA)
            maxA=A(j,i);
        end%{if}
        if ~isnan(A(j,i)) & (A(j,i)<minA)
            minA=A(j,i);
        end%{if}
    end%{for j}

    % rng stores ranges of every variable
    rng(i)=maxA-minA;
    if rng(i)==0
        msg='all elements in column i are same'
        rng(i)=1;
        % we want to avoid division by zero in case
        % all elements in column i are same; so one is
        % assigned to rng(i) instead of zero, summand in
        % formula for Gower distance is zero anyway
    end%{if}
end%{for i}

% we find Gower distance for every record; the result is a m
% times m matrix D; if at least one element in corresponding
% column is absent, this column gives 0 to total sum;
% for every record i from 1 to m, we find it's distance with
% records from i+1 to m (D is symmetric matrix, that is, i
% distance for with 1 to i-1 records already found)

for h=1:m
    for i=h:m
        s=0;
        % find weight of each variable i to total sum
        for j=1:n
            if ~isnan(A(h,j)) & ~isnan(A(i,j))
                s=s+abs(A(h,j)-A(i,j))/rng(j);
            end%{if}
        end %{for j}
        D(h,i)=s/n;
        % D symmetric matrix
        D(i,h)=D(h,i);
```

```
    end%{for i}
end%{for h}

% we determine for each record it's k neighbors; during calculation
    of
% minimum distance the first search gives us zero (distance
% with itself is zero) and we must take it into account;
% if we find that distance of record with different record is zero
% it means they coincide in each element in corresponding columns
% (they may not coincide in columns where records have missing
% values), we do not consider such records as neighbor and we
% find as neighbor record with which distance is greater than zero

% we use new matrix C instead of D because during search
% we have to change certain values of D;
C=D;
% we write record's own index in the first row IND matrix;
% distance with itself is zero
for j=1:m
        KN(1,j)=D(j,j);
        IND(1,j)=j;
end%{for j}

% elements of matrix C which are zeros, we change
% to one (maximum possible value)
for j=1:m
    C(j,j)=1;
end%{for j}

% we find for every record it's k least positive distances
% (ignore coinciding records)
% x represents i column of C
for j=1:m

   for i=1:m
      x(i)=C(j,i);
  end%{for i}

  % find min in x and minimum's index in x
  [P,I]=min(x);
  s=0;
  i=2;
  % find k minimums; if distance is zero, we ignore this
  % record and do process again until we find record with
  % minimum positive (not zero) distance; that is, distance
  % between any two records from group of records which consists
  % of initial record and it's k neighbors is always positive
  while (i<k+2)
      if P(1)==0
          t=I(1);
          x(t)=1;
      else
          KN(i,j)=s+P(1);
          IND(i,j)=I(1);
          t=I(1);
          x(t)=1;
          i=i+1;
```

```
            for r=1:m
                x(r)=x(r)-P(1);
            end%{for}
            s=s+P(1);
        end%{if else}
        [P,I]=min(x);
    end%{while}
end%{for j}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function S=Simulation1(A,IND,KN,m,n,k)
%Input - A is m times n matrix containing data set (every row is
%            an record containing n different variables)
%        m is number of rows in A (number of records)
%        n is number of columns in A (number of variables in
%            every record)
%        k is number of neighbors
%        IND is k+1 times m matrix containing indexes of k
%            closest neighbors in A matrix; the first row
%            of IND contains record's own index
%        KN  is k+1 times m matrix containing distance values
%               of k least distant records
%Output - S is m times m matrix containing simulated records; all
%            records are distinct from those records on which
%            bases they were simulated; but if it has been
%            impossible to get distinct simulated record after
%            certain number of iterations some modification is
%            used to resolve the problem

% random variable takes values 1,2,..,k+1 with equal
% probabilities 1 over k+1.

% determine probabilities p
str='equal';

for i=1:k+1
    p(i)=1/(k+1);
end

for j=1:m
    prob(j,:)=p;
end%{for j}

% the function below does simulations given p
[S]=SimulationMainBlock(A,IND,KN,m,n,k,prob,str);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [S]=Simulation2(A,IND,KN,m,n,k)
%Input - A is m times n matrix containing data set (every row is
%            an record containing n different variables)
%        m is number of rows in A (number of records)
%        n is number of columns in A (number of variables in
%            every record)
%        k is number of neighbors
%        IND is k+1 times m matrix containing indexes of k
```

```
%              closest neighbors in A matrix; the first row
%              of IND contains record's own index
%         KN   is k+1 times m matrix containing distance values
%                 of k least distant records
%Output - S is m times m matrix containing simulated records; all
%               records are distinct from those records on which
%               bases they were simulated; but if it has been
%               impossible to get distinct simulated record after
%               certain number of iterations some modification is
%               used to resolve the problem

% random variable takes values 1,2,..,k+1 with decreasing
% probabilities (more distant gets less probability); probability
% weights are 1/(s),1/(2s),....,1/((k+1)s) where s is chosen
% such that all probabilities sum to 1;

% determine probabilities p
str='decreasing';

s=0;
for i=1:k+1
    s=s+1/i;
end

for i=1:k+1
    p(i)=1/(i*s);
end

for j=1:m
    prob(j,1:k+1)=p;
end%{for j}

% the function does simulation given p
[S]=SimulationMainBlock(A,IND,KN,m,n,k,prob,str);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [S]=Simulation5(A,IND,KN,m,n,k)
%Input -  A is m times n matrix containing data set (every row is
%              an record containing n different variables)
%         m is number of rows in A (number of records)
%         n is number of columns in A (number of variables in
%              every record)
%         k is number of neighbors
%         IND is k+1 times m matrix containing indexes of k
%              closest neighbors in A matrix; the first row
%              of IND contains record's own index
%         KN   is k+1 times m matrix containing distance values
%                 of k least distant records
%Output - S is m times m matrix containing simulated records; all
%               records are distinct from those records on which
%               bases they were simulated; but if it has been
%               impossible to get distinct simulated record after
%               certain number of iterations some modification is
%               used to resolve the problem

% random variable takes values 1,2,..,k+1 with
```

```
% probabilities that depend on distance (more distant gets
% less probability); probability weights are (s-d1)/s, (s-d2)/s,
% ...., (s-dk)/s where s is chosen such that all probabilities
% sum to 1 and d1<d2<..<dk.

% determine probabilities p
str='distance';

for j=1:m
      c=true;
      key=k;
      while c
              b=false;
              s=0;

              % write probabilities for every record j in matrix
              % p(i,j); provide that sum of probabilities is one
              % and all not negative
              for i=2:key+1
                   s=s+KN(i,j);
              end%{if}
              s=s/(key-1);

              for i=1:key
                   prob(j,i)=(s-KN(i+1,j))/s;
                            end%{for i}

              % check if all probabilities are not negative
              for i=1:key
                        if prob(j,i)<0
                             b=true;
                        end%{if}
              end%{for}

              % if there are negative probabilities, we equate p(key)
              %  to zero and repeat process for key=key-1;
              if ~b
                   c=false;
              else
                   prob(j,key)=0;
                   key=key-1;
              end%{if}
       end%{while c}
end%{for j}

for j=1:m
     prob(j,k+1)=0;
end%{for j}

% the function does simulation given p
[S]=SimulationMainBlock(A,IND,KN,m,n,k,prob,str);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [S]=Simulation6(A,IND,KN,m,n,k)
%Input -  A is m times n matrix containing data set (every row is
```

```
%                an record containing n different variables)
%          m is number of rows in A (number of records)
%          n is number of columns in A (number of variables in
%                every record)
%          k is number of neighbors
%          IND is k+1 times m matrix containing indexes of k
%               closest neighbors in A matrix; the first row
%               of IND contains record's own index
%          KN  is k+1 times m matrix containing distance values
%                  of k least distant records
%Output - S is m times m matrix containing simulated records; all
%                records are distinct from those records on which
%                bases they were simulated; but if it has been
%                impossible to get distinct simulated record after
%                certain number of iterations some modification is
%                used to resolve the problem

% random variable takes values 1,2,..,k+1 with increasing
% probabilities (more distant gets more probability weight);
% probability weights are 1/((k+1)*s),1/(k*s),....,1/(1*s)
% where s is chosen such that all probabilities sum to 1.

% determine probabilities p
str='increasing';

s=0;
for i=1:k+1
    s=s+1/i;
end

for i=1:k+1
    p(i)=1/((k+2-i)*s);
end

for j=1:m
    prob(j,1:k+1)=p;
end%{for j}

% the function does simulation given p
[S]=SimulationMainBlock(A,IND,KN,m,n,k,prob,str);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [S]=SimulationMainBlock(A,IND,KN,m,n,k,prob,str);
%Input -  A is m times n matrix containing data set (every row is
%                an record containing n different variables)
%          m is number of rows in A (number of records)
%          n is number of columns in A (number of variables in
%                every record)
%          k is number of neighbors
%          IND is k+1 times m matrix containing indexes of k
%               closest neighbors in A matrix; the first row
%               of IND contains record's own index
%          KN  is k+1 times m matrix containing distance values
%                  of k least distant records
%            prob is m times k+1 matrix of probabilities
```

```
%           str is a string that denotes choice of probabilities
%Output - S is m times m matrix containing simulated records; all
%               records are distinct from those records on which
%               bases they were simulated; but if it has been
%               impossible to get distinct simulated record after
%               certain number of iterations some modification is
%               used to resolve the problem

% we consider j record (from 1 to m) and its k neighbors;
% we do n independent draws of rv for every of n columns.
% every time we get some realization of rv equal to i, we
% take element from i record from that column.
% after we get simulated record, we check if it coincides with any
% record from original matrix A, if it does we redo the process
% again; if after 10000 tries, we can't get distinct record, we
% use modification to process of simulation. namely, we add
% to simulation elements of the next closest neighbor; so we
% consider k+1 neighbors for j record instead of k; we continue
% simulation for j record and if, again, we don't get distinct
% record after 10000 tries, we add again the next neighbor
% and so on.

% for every record from 1 to m
for j=1:m
    c=true;
    g=0;
    e=0;

    p=prob(j,:);
    % do "while" until simulated record is different from those in
    % matrix A. every time number of iterations becomes greater
    % than certain level, we increase number of neighbors for
    % that record by one
    while c
        % simulation of n variables for j record;
        for i=1:n
            x=rand(1);
            t=0;
            h=1;
            b=true;
            s=p(1);

            % variable t takes values from 1 to k+1 with
            % given probabilities
            while b
                t=t+1;
                if (s>x)
                    b=false;
                else
                    h=h+1;
                    s=s+p(h);
                end%{if else}
            end%{while b}
            index=IND(t,j);
            B(1,i)=A(index,i);
        end%{for i}
```

```
            % we got simulated record and we check if there is such
            % record in matrix A; b equals true if it is distinct
            [b]=Check2(A,B,m,n);
            if b
                 c=false;
                 S(j:j,1:42)=B(1:1,1:42);
            end%{if}

            g=g+1;
            if (g>10000)
                   % if after 10000 iterations, we can't get distinct
                   % record, we do modification to simulation process
                   % by increasing the number of neighbors by one;
                   % function RandFromCol does it
                   e=e+1;
                   [p]=ReAssignProb(j,A,IND,KN,n,k,e,str,p);
                   g=0;
                   j
            end%{if}
     end%{while c}
end %(for j)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [b]=Check2(A,B,m,n)
%Input - A is m times n matrix containing data set (every row
%                 is an record containing n different values)
%        m is the number of rows in A (number of records)
%        n is the number of columns in A (number of elements in
%                every record)
%        B is a simulated record (row vector)
%Output - b is boolean variable that equals
%           false if we have the identical record in matrix A;
%           otherwise, it is true

% compare B with every record in matrix A
b=true;
k=1;
while (k<m+1) & (b)
                  c=true;
                  i=1;
                  % loop through every record of matrix A
                  while (i<n+1) & (c)
                            if B(1,i)~=A(k,i)
                                 % c becomes false if B and A(k,i) are
                                 % distinct
                                 c=false;
                            end %{if}
                            i=i+1;
                  end %(while i)
                  % c is true if we have found identical record in A
                  if c
                       b=false;
                  end%{if}
                  k=k+1;
end%{while k}
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [p]=ReAssignProb(j,A,IND,KN,n,k,e,str,p);
%Input - A is m times n matrix containing data set (every row is
%            an record containing n different variables)
%        m is number of rows in A (number of records)
%        n is number of columns in A (number of variables in
%            every record)
%        k is number of neighbors
%        IND is k+1 times m matrix containing indexes of k
%            closest neighbors in A matrix; the first row
%            of IND contains record's own index
%         j is an index of current record in matrix A on a base
%             of which simulation occurred
%        KN  is k+1 times m matrix containing distance values
%             of k least distant records
%         p is m times k+1 matrix of probabilities
%         str is a string that denotes choice of probabilities
%              of which simulation occurred
%         e is integer that shows how many additional
%                   neighbors have been added
%Output - B is 1 times m simulated record;

% increase number of neighbors from k to d=k+e
d=k+e;

% determine probabilities p for every of four possible values
% of str
switch str
      case 'equal'
            for i=1:d+1
                  p(i)=1/(d+1);
            end%{for}
      case 'decreasing'
            s=0;
            for i=1:d+1
                 s=s+1/i;
            end%{for}

            for i=1:d+1
                  p(i)=1/(i*s);
            end%{for}
      case 'increasing'
            s=0;
            for i=1:d+1
                 s=s+1/i;
            end%{for}

            for i=1:d+1
                  p(i)=1/((k+2-i)*s);
            end%{for}

      case 'distance'
            c=true;
            key=d;
```

```
            while c
                  b=false;
                  s=0;
                  % assign probabilities for record j in vector
                  % p(i); provide that sum of probabilities is
                  % one and all not negative
                  for i=2:key+1
                        s=s+KN(i,j);
                  end%{for}
                  s=s/(key-1);

                  for i=1:key
                        p(i)=(s-KN(i+1,j))/s;
                   end%{for i}

                  % check if all probabilities are not negative
                  for i=1:key
                        if p(j,i)<0
                              b=true;
                        end%{if}
                   end%{for}

                  % if there are negative probabilities, we
                  % equate p(key) to zero and repeat process
                  % for key=key-1;
                  if ~b % not b
                       c=false;
                  else
                       p(key)=0;
                       key=key-1;
                  end%{if}
            end%{while c}

      case 'optimal'
            for i=2:d+1
                 p(i)=(1-p(1))/(d);
            end%{for}
```

end%{switch}

The running time to produce data set S with $m = 2972$ records by this method is approximately 530 seconds. The result of simulations is very good, in sense, that we get data set S, which eigenvalues of correlation matrix are close to original. As an example, the eigenvalues of some simulated matrix using $p_0 = 0.9$, $p_1 = 1 - p_0 = 0.1$ and $k = 1$ are shown in the following tables. For this matrix S, the absolute value of difference between corresponding loadings is greater than 0.1 for 0.091% of all loadings and is greater than 0.5 for 0.011% of all loadings.

13.11	3.53	2.40	1.87	1.65	1.37	1.53	1.16	1.05

Table 3: The first 9 largest eigenvalues of correlation matrix of data set S simulated by k-neighborhood method when $k = 1$ and $p = 0.9$

$\mathbf{F_1}$	$\mathbf{F_2}$	$\mathbf{F_3}$	$\mathbf{F_4}$	$\mathbf{F_5}$	$\mathbf{F_6}$	$\mathbf{F_7}$	$\mathbf{F_8}$	$\mathbf{F_9}$
0	0	-0.01	-0.01	0.04	-0.38	0.08	0.04	0.04
0	0.02	-0.01	0	-0.04	-0.11	-0.01	-0.07	0
0	0.01	0.01	-0.01	-0.01	-0.2	-0.01	0	-0.02
0	0	0	0	-0.01	0.33	-0.01	0	0
0.01	-0.02	-0.02	0.04	0.05	-1.04	0.09	-0.04	-0.01
0	-0.01	-0.02	0.06	0.06	-0.96	0.08	-0.03	0
0	0.01	-0.01	-0.02	-0.03	0.54	-0.02	-0.02	-0.01
0.01	-0.01	-0.01	-0.02	-0.03	0.47	-0.02	-0.01	0.01
0	0	-0.02	-0.02	-0.03	0.48	-0.01	-0.02	0.01
0	-0.01	0	0.01	-0.02	0.45	0	-0.03	0.01
0	-0.01	0.01	0.01	-0.02	0.09	-0.01	0	0.01
0.01	0	0.02	0.02	-0.02	0.06	-0.04	-0.01	-0.01
0	-0.01	0	0.02	-0.03	0.3	-0.03	0	0.01
0	0.01	0	0	-0.01	-0.23	0	0.01	0.01
0	0.01	-0.01	-0.01	0	0.18	0.01	0.01	0.01
0	0	0	-0.01	-0.01	0.29	0.01	0.01	0.02
0.01	0	0	0	0	0.06	-0.01	-0.02	0.04
0.01	-0.01	0	-0.02	0	0.03	0.01	-0.01	0.03
0	-0.01	0	-0.01	0.02	-0.19	0.01	0	0.03
0.01	0	0	-0.01	0.02	-0.39	0.02	-0.01	0.01
0	0	0	-0.01	0.03	-0.32	0.03	0.01	-0.03
0.02	0	0.01	0.01	0.02	-0.19	0	0	-0.03
0.01	0.02	0	0.01	0.04	-0.47	0.02	0.04	-0.05
-0.01	0	-0.01	0.01	0.01	0.15	0	0.03	-0.02
0	0	0	-0.02	0	0.21	-0.01	-0.03	0.03
0	0	0.01	0.01	0	0.01	0	0.01	-0.02
0.01	0	-0.01	0	0.01	-0.33	0.01	0.02	0
0.01	0	-0.02	0.03	0.02	-0.41	0.02	0.02	-0.01
0.01	0	0	-0.01	0.01	-0.22	0.01	0.01	-0.03
-0.02	0	0.01	0	-0.02	0.29	-0.06	-0.01	0
0	0.01	-0.02	0	-0.01	0.12	-0.03	-0.01	0.02
0.01	-0.01	0	0.02	0.04	-0.33	0.02	-0.02	0.03
0.01	0.01	-0.01	0	0.01	-0.06	-0.01	0.01	0
0.01	-0.01	0.01	0.01	0.06	-0.37	0.03	-0.01	-0.01
0.01	0	0	0	0.01	-0.07	0	0.01	-0.01
0	0.02	0	-0.01	0	0.16	-0.01	0.01	0
0.01	-0.01	0.01	0.03	0.06	-0.4	0.01	0	0.01
0	0.01	0	-0.02	-0.03	0.5	-0.01	0	0.01
0	0.02	0.02	0.03	0	-0.31	-0.02	0.02	-0.03
0.01	0	0.01	-0.01	0.04	-0.37	0.02	0	0
0	0.01	-0.01	0	0.02	-0.26	0	0.01	-0.01
0	0	0	-0.01	-0.08	0.88	-0.07	-0.04	-0.01

Table 4: The matrix of differences between corresponding factor loadings of S and A (S is simulated by k-neighborhood method when $k = 1$ and $p = 0.9$)

The simulated matrix S is different from the original in sense, that every record from 1 to m might have different neighbors, that is, indexes of neighbors are distinct and values of k least distances are also distinct. But if we do factor analysis of original and simulated matrices using principal component method and as a criteria for choosing the number of factors we choose the number of eigenvalues that have value greater than one, the initial matrix gives 9 factors and the simulated matrix gives 8 or 9 in dependence of simulated data set. But the eigenvalues are very close to each other. The proportions that these 9 eigenvalues is account for are in both case relative close.

The following optimization problems are the natural consequence: what value of k and what probability weights minimize MSE of eigenvalues and factor loadings, if as a true values of eigenvalues and factor loadings, we use those values of original data set A,

$$\min_{\mathbf{k},\mathbf{p_0},..,\mathbf{p_k}} \mathbf{MSE}_\lambda = \sum_{\mathrm{i}=1}^{\mathrm{n}} \mathbf{E}(\lambda_{\mathrm{i}}^{\mathrm{A}} - \lambda_{\mathrm{i}}^{\mathrm{S}})^2 = \mathbf{Var}_\lambda + \mathbf{Bias^2}$$

$$\min_{\mathbf{k},\mathbf{p_0},..,\mathbf{p_k}} \mathbf{MSE}_l = \sum_{\mathrm{i}=1}^{\mathrm{n}} \sum_{\mathrm{j}=1}^{\mathrm{r}} \mathbf{E}(l_{\mathrm{ij}}^{\mathrm{A}} - l_{\mathrm{ij}}^{\mathrm{S}})^2 = \mathbf{Var}_l + \mathbf{Bias^2}$$

where n is the number of variables and r is the number of factors. I do not think it is possible to find solution in closed form, so I used simulations to estimate mean square error in both cases. I simulated data using different probability weights and different k, in every case I simulated data set 100 times to get estimate of MSE. As probability weights I considered four cases: equal, increasing, decreasing and distance dependent. So I used four different types of weights that a user can choose from the pop-up menu using the application. As k, number of neighbors, I chose k equal to 1, 5 and 10. The following two tables show the results I got from simulations. One table is for MSE of eigenvalues, the second is for MSE of factor loadings when number of factors is equal to 9.

MSE of eigenvalues	k=1	k=5	k=10
Equal	1,81	4,53	5,47
Decreasing	0,77	2,29	2,97
Increasing	3,53	5,90	6,31
Distance related	1,65	3,73	4,05

Table 5: MSE of eigenvalues of simulated data set S by k-neighborhood method with given k and probabilities weights

MSE of factor loadings	k=1	k=5	k=10
Equal	3,22	9,59	11,46
Decreasing	1,24	7,05	8,65
Increasing	4,65	10,28	11,54
Distance related	3,47	8,80	9,22

Table 6: MSE of factor loadings of data set S simulated by k-neighborhood method with given k and probabilities weights (number of factors is 9)

The result tells that it looks like that the optimal choice of k is one and decreasing choice of probability weights is more preferable. Then for k equal to 1, I chose different probability weights. The following table shows resulting estimates (to get an estimate, I did 100 simulations of data set). The first column of the table is probability p assigned to initial record and $1 - p$ is probability assigned for a neighbor.

Probability p	MSE_λ	MSE_l
0.60	1.12	2.12
0.70	0.63	0.95
0.80	0.30	0.478
0.90	0.10	0.189
0.95	0.05	0.112
0.99	0.03	0.07

Table 7: MSE of eigenvalues and factor loadings of data set S simulated by k-neighborhood method when $k = 1$ and with given p

So the results tell us that we should choose probability that we assign to the initial record close to one and one minus the first probability to a neighbor record. Of course, probability should not be equal to one; in this case the program shows error message because it is not possible to simulate records distinct from records of data set A under this choice of probabilities. The disadvantage of a choice of probability close to one is more working time of code and more iterations. In the application, the button called *Optimal k neighborhood* simulates data set using $k = 1$ and a user has to fill only desirable probability weight. The function InputNum asks a user to input probability value and checks if the inputted value is less than one and is greater than zero. The code behind the button and all embedded functions that where not appeared before is shown below.

```
% --- Executes on button press in pushbutton6.
function pushbutton6_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton6 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

```
% input probability and check if it is less than one and greater
% than zero
str='Enter p value:';
type='real';
answer=InputNum(str,0,1,type);
if ~isempty(answer) & (answer~=-1)
              % starting initializations
               A=handles.current_data;
               n=handles.n;
               m=handles.m;
               lamda=handles.lamda;
               dmissing=handles.dmissing;
               p1=answer;

               % we calculate distances and determine neighbors for
               % all records of A
               [D,IND,KN]=GowerDistance(A,m,n,200);

               % the result of simulation is matrix S
               [S]=SimulationP(A,IND,KN,m,n,p1);

               % check if S has missing elements
               if dmissing
                    dmissing=false;
                     for i=1:m
                            for j=1:n
                                 if isnan(S(i,j))
                                      dmissing=true;
                                  end%{if}
                             end%{for j}
                    end%{for i}
               end%{if dmissing}

               % result of simulation is matrix S
               assignin('base','S',S);
               % plot eigenvalues and squared error
               [lam,error]=PlotEigErr(S,lamda,dmissing,n,'black','--
                    bo');
               % calculated error is shown on application form
               set(handles.text16,'String',error);
               msgbox('Output is matrix S');
end%{if ~isempty(answer)}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [S]=SimulationP(A,IND,KN,m,n,p1);
%Input -  A is m times n matrix containing data set (every row is
%              an record containing n different variables)
%         m is number of rows in A (number of records)
%         n is number of columns in A (number of variables in
%              every record)
%         p1 is probability assigned to initial record
%         IND is k+1 times m matrix containing indexes of k
%              closest neighbors in A matrix; the first row
%              of IND contains record's own index
%         KN  is k+1 times m matrix containing distance values
```

```
%                 of k least distant records
%Output - S is m times m matrix containing simulated records; all
%               records are distinct from those records on which
%               bases they were simulated; but if it has been
%               impossible to get distinct simulated record after
%               certain number of iterations some modification is
%               used to resolve the problem

% our random variable takes two values 1 and 2 with
% probabilities p1 assigned to initial record and 1-p1
% assigned to the first neighbor;

% determine probabilities p
str='optimal';

p(1)=p1;
p(2)=1-p1;

for j=1:m
    prob(j,:)=p;
end%{for j}

% the function does simulation given p
[S]=SimulationMainBlock(A,IND,KN,m,n,1,prob,str);
```

3 The second method: relative frequency

We simulate new data set using nonparametric approach. The simulation process of new record is the following: we randomly select one column of matrix A. Each column has equal probability to be chosen. After we have selected a column, we randomly select an element in that column. The probabilities assigned to elements are estimated by relative frequencies of those elements. The new simulated record has the chosen element in chosen column. On the next step, we select one of the remaining columns and then we randomly select an element in that column where again probabilities are equal to relative frequencies of that column's elements. And so on until we fill all variables of new record. If we do this process using independent draws, that is, we select element in column independently of previously chosen elements, we don't reach our aim to get data set that resembles data set A in factor analysis sense (namely, if we perform factor analysis on simulated data set, we get all eigenvalues close to one due to independence). That means, we have to try to preserve somehow dependence between elements. We try to use conditional probabilities where conditions are made on k last chosen elements. The formula for estimating conditional probabilities is shown below.

$$P(\xi_{j_{k+1}} = y_{k+1} | \xi_{j_k} = y_k, \cdots, \xi_{j_1} = y_1) = \frac{\#\text{ records where } \xi_{j_{k+1}} = y_{k+1} \text{ given history}}{\text{total number of records given history}}$$

where $j_{k+1}, \cdots, j_1$ denote indexes of columns of matrix A chosen on steps $1, 2, .., k, k+1$.

We choose parameter k, where k is how many past elements to be taken into account (if we take $k = 1$, it is analogue to Markov chain). On every step of simulation process described above, we have randomly chosen some k past elements in k randomly chosen columns (initially, it is null set or set that contains fewer than k elements). We select from matrix A records that have same values as these k given elements in k given columns and we estimate probabilities using this selected part of matrix A (initially, we use all matrix A) by calculating relative frequencies of elements encountered in this selected part of A. So on every step of simulation process, we first select column and then element in that column, but use for randomization probabilities equal to estimates of conditional probabilities given k past values. The code written to realize this simulation method accounts for possibility of missing data. If we have missing values in some column of selected part of matrix A, we just count them as count usual numbers and we assign to probability, that we get missing value, number equal to calculated frequency divided by total number of elements in that column.

The button called *Relative frequency* on the application form simulates data set using this method. When a user pushes the button, dialog box appears where a user has to input desirable number on how many past elements conditional probability depend. The function *InputNum* does that and, in addition, it verifies if an integer positive number has been inputted; otherwise, error message appears. Then the function *Simulation9* is executed. The main part of the body of the program contains loop 'while' that fills step by step every variable of new simulated record. The loop calls several functions on each step. First, the function *DiscreteRand1* selects randomly one of not previously chosen columns. Every column has equal probability to be chosen. Then the function *RetFreq* finds all distinct elements in columns of given matrix B and finds frequencies of these distinct elements. The given matrix B is some part of matrix A where we select records from A that have in certain k columns certain k elements and, initially, it is full matrix A. In addition, if matrix has missing elements, the function counts them and writes frequencies in the first row of output matrix F; if there are not missing elements, matrix F contains zeros in the first row. Then the function *DiscreteRand2* selects randomly one of elements in the column chosen by the function *DiscretRand1*, where probabilities are estimated by relative frequencies found by the function *RetFreq*. The last function *UpdateBgH* updates given matrix B taking into account that we have now different k past elements by selecting only records from matrix A, that have new k elements in new k columns. (In fact, only one element and one column have been changed.) The loop terminates when we fill every variable of new record. Then the function *Check2* verifies if a simulated record is distinct from records of data set A. The result of execution of the function *Simulation9* is matrix of simulated records S and it is stored in Matlab's workspace by S name. In addition, square error (sum of squared differences of corresponding eigenvalues of correlation matrices of A and S) is calculated and all eigenvalues of correlation matrix of S are illustrated on the application form by the function *PlotEigErr*. During execution of this method, we only have to deal with discrete random variables and they are simulated on a base of uniform random variable which is

generated by random generator of Matlab.

The running time of simulations is large: simulation of 100 records given k lasts approximately 3000 seconds. The method doesn't provide desirable result. I simulated 100 records for $k = 1$, $k = 5$, $k = 10$. The eigenvalues are presented in the following table.

k=1	3.12	2.63	2.37	2.18	1.92	1.74	1.68	1.64	1.61
k=5	3.47	2.30	2.23	2.03	1.94	1.92	1.84	1.67	1.61
k=10	3.90	2.37	2.16	1.98	1.91	1.8	1.75	1.62	1.53

"The first 9 largest eigenvalues of correlation matrix of data set S simulated by relative frequency method using given k"

So even for large k, the eigenvalues get maximum value of 4; in case of original data set, the largest eigenvalue is equal to 12.92. The square error of eigenvalues for $k = 5$ is 94.87 and that of factor loadings is 29.59. The code behind the button and all embedded functions with comments is shown below.

```
% --- Executes on button press in pushbutton5.
function pushbutton5_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton5 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% a user should choose desirable number of past elements
str='Enter the number on how many past elements current state
    probabilities depend:';
type='integer';
answer=InputNum(str,0,handles.n,type);
if ~isempty(answer) & (answer~=-1)
                % starting initializations
                A=handles.current_data;
                n=handles.n;
                m=handles.m;
                lamda=handles.lamda;
                dmissing=handles.dmissing;
                k=answer;

                % the result of simulation is matrix S
                [S]=Simulation9(A,m,n,k);

                % check if S has missing elements
                if dmissing
                    dmissing=false;
                    for i=1:m
                        for j=1:n
                            if isnan(S(i,j))
                                dmissing=true;
                            end%{if}
                        end%{for j}
                    end%{for i}
```

```
                end%{if dmissing}

                % result of simulation is matrix S
                assignin('base','S',S);
                % plot eigenvalues and squared error
                [lam,error]=PlotEigErr(S,lamda,dmissing,n,'black','--
                    bo');
                % calculated error is shown on application form
                set(handles.text16,'String',error);
                msgbox('Output is matrix S');
else
                msgbox('input must be an integer number in certain
                    range');
end%{if ~isempty(answer)}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [S]=Simulation9(A,m,n,k);
%Input - A is m times n matrix containing data set;
%        m is the number of rows in A (number of records)
%        n is the number of columns in A (number of
%           variables of every record)
%        k is the number of past elements that will be taken
%          into account while calculating probabilities (history)
%Output - S is m times n matrix containing simulated records; all
%           records of S are distinct from records

tic;
% key shows current number of simulated records
key=1;
while key<101%m+1 %simulate m records

        % CONDB is an array consisting of n elements. each element
        % is a boolean variable that takes value true if column of
        % A with corresponding index was already randomly chosen
        % on previous steps.
        % CONDV is an array consisting of n elements. each element
        % is a rational number that takes value of randomly
        % selected element. index of element in array corresponds
        % to index of column in matrix A from where this element
        % was chosen. SSS is a simulated record

        % initial initialization of CONDB, CONDV, SSS as false, NaN
        % and NaN
        for i=1:n
                CONDB(1,i)=false;
                CONDV(1,i)=NaN;
                SSS(1,i)=NaN;
                end%{for i}

        % history tracks previously chosen columns and elements in
        % those columns; initial initialization: first row shows
        % indexes of columns in matrix A; the second one shows
        % what elements were chosen in those columns; the column
        % that was chosen on the last step has index 1; the column
        % that was chosen before the last has index 2 and so on.
```

```
for i=1:k
        history(1,i)=NaN;
        history(2,i)=NaN;
        end%{for i}

%i initially B=A
B=A;
rowB=size(B,1);
boolean=true;

i=n;
while (i>0)&(boolean)
        % we randomly select one of not previously chosen
        % column of matrix B; every remaining column can
        % be chosen with equal probability;
        % after column's choice, we update array CONDB:
        % CONDB has one less free column t is an index
        % of chosen column
        [t,CONDB]=DiscreteRand1(i,CONDB,n);

        % given chosen t, we update the first row of
            history;
        % the first element of matrix "history" shows the
        % last chosen column
        for j=1:k-1
                history(1,j+1)=history(1,j);
        end%{for j}
        history(1,1)=t;

        % we find for each column of matrix B what values
        % each column can take and we count how many each
        % element is presented in certain column of matrix
        % B; matrix V contains distinct values; matrix F
        % contains frequencies corresponding to values in
        % matrix V
        F=0;
        V=0;
        [F,V]=RetFreq(B,rowB,n);
        rowF=size(F,1);
        max=rowF+1;

        % randomly select one element in chosen column t
        % of matrix B; probabilities of selection are
        % estimated by relative frequencies; all possible
        % values are in matrix V and all frequencies are
        % in matrix F; val is a value of chosen element in
        % column t; after selection, we update matrix
        % CONDV which reflects all chosen elements
        [val,CONDV]=DiscreteRand2(t,rowB,max,CONDV,F,V);

        % given chosen val, we update the second row
        % of history; the first element of matrix "history"
        % shows the last chosen element
        for j=1:k-1
                history(2,j+1)=history(2,j);
        end%{for i}
        history(2,1)=val;
```

```
            % we got simulated value for a column t of our
            % new record
            SSS(1,t)=val;

            % we update B in accordance with the last k chosen
            % elements; we extract from matrix A records using
            % the following principle: the resulting matrix
            % has only records where values on k last
            % chosen columns are equal to k last chosen
            % elements (matrix history contains all information
            % on the k last chosen columns and values)
            [B]=UpdateBgH(A,m,n,history);
            rowB=size(B,1);

            % if rowB is one, we have only one record in B and
            % there is no sense to continue process
            if rowB==1
                boolean=false;
                SSS=B;
            end%{if}
            i=i-1;

      end%{while}


      % we check if simulated record coincides with any
      % record from initial data set A; if it does we repeat
      % again the process again;
      [c]=Check2(A,SSS,m,n);
         if c
            for k=1:n
                S(key,k)=SSS(1,k);
            end%{for k}
            key=key+1;
            toc;
        end %{if}
       clear SSS;

end%{while key}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [t,CONDB]=DiscreteRand1(i,CONDB,n)
%Input - i is the total number of not already chosen columns
%            all columns have equal probability to be chosen
%        CONDB is an array which elements take value true if column
%                with same index was already chosen
%         n is a number of columns of matrix A
%Output - t is an index of randomly chosen column
%     CONDB is an array which elements take value true if column
%            was already chosen; after choice of new column we update
%            CONDB (we have one less free column)

% we have i free columns and we choose randomly with equal
% probabilities one of the columns; probability is 1 over i;
```

```
% we update array CONDB as we have one less free column
x=rand(1);
b=true;
t=1;
p=1/i;
while b
        if  (p>x)
                b=false;
        else
                p=p+(1/i);
                t=t+1;
        end%{if else}
end%{while b}

c=true;
k=0;
j=1;
while (k<t)
         if CONDB(1,j)==false
                k=k+1;
         end%{if}
         j=j+1;
end%{while isnan}
j=j-1;
CONDB(1,j)=true;
t=j;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [val,CONDV]=DiscreteRand2(t,rowB,max,CONDV,F,V)
%Input - t is an index of randomly chosen column;
%        rowB is the number of rows in matrix B;
%        max is 1 plus the number of rows of matrix F
%        CONDV is an array that shows what concrete values in
%                chosen columns were selected on previous steps
%        V is h times n matrix containing different distinct
%            values of elements encountered in certain column of A
%        F is h times n matrix containing frequencies of every
%           distinct element in certain column of A; the concrete
%           value of  h is determined during execution of code
%Output - val  is a chosen random value in column t
%        CONDV is an array that shows what elements were chosen
%                previously; after val has been randomly chosen in
%                column we update array CONDV by adding new choice

% the first row of matrix F is reserved to show number of missing
% elements in every column; if there is no missing elements, then
% we assign value 2 to k for column t; otherwise k=1; k shows
% from where we have to look at matrix F
   if F(1,t)==0
         k=2;
        else
         k=1;
        end%{if}
% we want to know how many distinct elements has column t; value
% h shows it; the first time we meet zero or we go further than
```

```
% row length of F, we determine value of h
    h=k;
        while  (h<max) & (F(h,t)~=0)
              h=h+1;
        end%{while}
        h=h-1;

% we estimate probabilities by relative frequencies
% (frequency/total number); matrix F contains frequency of
% every element
    i=0;
        s=0;
        for j=k:h
            i=i+1;
            s=s+(F(j,t)/rowB);
            p(i)=s;
        end%{for j}

% we simulate element in column t of new record using our
% estimation of probabilities
        x=rand(1);
        b=true;
    i=1;
    while b
            if  (p(i)>x)
                  b=false;
            else
                  i=i+1;
            end%{if else}
        end%{while b}

 % if k=2, it means we started from the second place in matrix F,
 % so we have elements from 2 to h; but index of vector of
 % probabilities is from 1 to h-1; so taking this into account,
 % we have to change i on i-1 to select correct element
    if k==2
       i=i+1;
    end%{if}

 % we write the chosen element in column t in array CONDV on place
     t
    CONDV(1,t)=V(i,t);
        val=V(i,t);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [C]=UpdateBgH(A,m,n,history);
%Input - A is m times n matrix containing data set
%        m is the number of rows in A (number of records)
%        n is the number of columns in A (number of variables
%            of every record)
%        history is the matrix containing indexes of chosen
%        columns and values in corresponding columns
%Output - C is a matrix containing only records from A that
%               take certain values in certain columns; actual
%               indexes of last k selected columns and values in
```

```
%                 those columns are stored in matrix called history

% we update B in accordance with the last k chosen elements in
% certain columns; we extract from matrix A only records that
% have in the k last chosen columns same values as k last
% chosen elements

% k checks of how many elements already in history (k can't be
% greater than value of col)
col=size(history,2);
k=1;
while (k<col+1)& (~isnan(history(1,k)))
    k=k+1;
end%{while}

k=k-1;
C=A;
% we extract from A only records that have in certain columns
% certain elements; "for loop" selects from A records that meet
% requirements given in array history
for i=1:k
                h=1;
        rowC=size(C,1);
        ind=history(1,i);
        val=history(2,i);

        % if val takes value NaN, it means that in that column was
        % randomly chosen missing element; so we select from
        % A those records that have missing values in the given
        % column
        if isnan(val)
                    for j=1:rowC
                            if isnan(C(j,ind))
                                    for k=1:n
                                            B(h,k)=C(j,k);
                                    end%{for k}
                                    h=h+1;
                            end%{if}
                    end%{for j}
         else % if val is not NaN, we select from A records that
                 % take value equal to val in given column
                     for j=1:rowC
                            if C(j,ind)==val
                                    for k=1:n
                                            B(h,k)=C(j,k);
                                    end%{for k}
                                    h=h+1;
                            end%{if}
                    end%{for j}
         end%{if}
         C=B;
         clear B;
 end%{for i}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
function [F,V]=RetFreq(A,m,n)
%Input - A is m times n matrix containing data set
%             m is number of rows in A
%             n is number of columns in A
%Output - V is h times n matrix containing different unequal
%                   values of elements encountered in certain
%                   column of A
%          F is h times n matrix containing frequencies of every
%            distinct element in certain column of A; the concrete
%            value of  h is determined during execution of code

% code goes through each column separately and finds how many
% times we have each unique element; every time we meet new
% element in certain column, we check if we had this element before
% by comparing with previous encountered elements; if we had
% this element before, we just increase frequency of this element
% in matrix F by one; if it is new element, we put the value of
% this element in matrix V and assign corresponding to this
% element frequency to one in matrix F

% using loop "for", we find the first not missing element in every
% column from 1 to n; we put this element on the second place in
% matrices F and V; first place in F and V is reserved to show
% frequency of missing elements
for i=1:n
    j=1;
    b=true;
    while (j<m+1) & (b)
          if isnan(A(j,i))
             j=j+1;
          else
             b=false;
             V(2,i)=A(j,i);
             F(2,i)=1;
          end%{if else}
    end%{while}
end%{for i}

% we use "for" loop to determine frequency of each element in each
% column separately for every column; vf is increased by one every
% time we encounter element that we have not had before
for i=1:n % for every column from 1 to n
    F(1,i)=0;
    V(1,i)=NaN;
    vf=1;
    for j=1:m % for each element in column i

        b=true;
        k=vf;
        while (k >0) & (b)

              if isnan(A(j,i))% check if the value is missing
                   F(1,i)=F(1,i)+1;
                   b=false;
              else % if value is not missing we check if we
                     % had this element before
              if   V(k,i) == A(j,i)
```

```
                    F(k,i)=F(k,i)+1;
                    b=false;
                end %{if}
            end%{if}
            k=k-1;
        end%{while}

        if b % b true only if we did not meet this element
             % before so we increase vf by one
            vf=vf+1;
            V(vf,i)=A(j,i);
            F(vf,i)=1;
        end%{if}

    end %{for j}
end%{for i}
```

4 The third method: basic model

The factor analysis given k factors assumes the following model:

$$x_i - \mu_i = l_{i1}\mathrm{F}_1 + l_{i2}\mathrm{F}_2 + \cdots + l_{ik}\mathrm{F}_\mathrm{k} + \epsilon_i,$$

where k is number of factors, μ_i is a mean of variable x_i $i = 1, .., n$, ϵ_i is specific term for variable x_i, $\mathrm{F_i}$ and ϵ_i are independent, $\mathbb{E}(\mathrm{F}) = 0$, $\mathsf{Cov}(\mathsf{F}) = \mathbb{I}$, $\mathbb{E}(\epsilon) = 0$, $\mathsf{Cov}(\epsilon) = \Psi$. We simulate data sets S using this model. We choose some number of factors k and some matrix of coefficients L. Then we simulate k independent standard normal random variables to get values for k factors and simulate n (number of variables) independent normal variables with mean equal to zero and variance equal to sample variance of corresponding variable x_i. So we plug all values into the model (μ is equal to sample mean of corresponding variables). The result of simulation is n values that can be integer, rational or irrational and positive or negative. But every of n random variables from original data set A is always positive and takes values from certain set of possible values $\mathbb{B}_i$ for $i = 1, .., n$. We want that simulation process provides us with n values, each from the corresponding set $\mathbb{B}_i$. So we have to transform n simulated values to some n values that are contained in the corresponding set of possible values $\mathbb{B}_i$ using some rule. To transform n values, we find the closest value in the corresponding set $\mathbb{B}_i$ for each of n values and replace simulated value by the found value from $\mathbb{B}_i$. After we get transformed n values, which compose simulated record, we check if there is such record in our original data set A. If there is no such record, the record is added to matrix S.

This model depends on choice of distribution for factors and for specific factors and two inputs: number of factors and matrix of loadings. If we use rule that tells to select number of factors equal to number of eigenvalues that greater than one, then the original data set A suggests to choose 9 factors. If we use 9 factors as input for the model, it is desirable to find a matrix L such

that we can get minimum possible value for MSE of factor loadings. Simulations has to be used to find such matrix L, but matrix L consists of $n \times 9$ elements and some of them in some dependence. So if we choose all elements arbitrarily independent each of other, it has to be performed very large number of simulations by randomness get desirable result. Instead for matrix L, we use the determined structure and this structure is suggested by the factor analysis applied to original data set A. We take matrix of loadings of data set A and select only that loadings that have enough large values, others we equate to zero. The table with factor loadings is shown on the following table.

$\mathbf{F_1}$	$\mathbf{F_2}$	$\mathbf{F_3}$	$\mathbf{F_4}$	$\mathbf{F_5}$	$\mathbf{F_6}$	$\mathbf{F_7}$	$\mathbf{F_8}$	$\mathbf{F_9}$
0	0	0	0	0	0.21	0	-0.53	0
0	0	0	0	0	0	0	0	-0.8
0	0.23	0	0	0	0	0	0.37	0.44
0.73	0	0	0	0	0	0	0	0
0	0	0	0.33	0	0.49	-0.58	0	0
0	0	0	0.26	0	0.44	-0.68	0	0
0.78	0	0	0	0	-0.28	0	0	0
0.77	0	0	0	0	-0.25	0	0	0
0.63	0	0	0	0	-0.26	0	0	0
0.74	0	0	0	0	-0.24	0	0	0
0.63	0	0	0	0	0	0	0	0
0.61	0	0	0	0.45	0	0	0	0
0.68	0	0	0	0.31	0	0	0	0
0.45	0	-0.25	0	0.42	0	0	0	0
0.7	0	0	0	0	0	0	-0.23	0
0.7	0	0	0	0	0	0	-0.25	0
0.63	0	0	0	0	0	0	0	0
0.64	0	0	0	0	0	0	0	0
0.62	0	-0.38	0	0	0	0	0	0
0.48	0	-0.55	0	-0.32	0.2	0	0.2	0
0.49	0	-0.53	0	0	0	0	0	0
0.63	0	0	0	0	0	0	0	0
0.41	0	-0.21	-0.21	0.37	0.26	0	0	0
0	-0.75	0	0	0	0	0	0	0
0	-0.7	0	0	0	0	0	0	0
0	-0.8	0	0	0	0	0	0	0
0.54	0	0.45	0.23	0	0	0.27	0.2	0
0.57	0	0.43	0	0	0	0	0	0
0.71	0	0.31	0	0	0	0	0	0
0	0.3	0	0	0	0	-0.21	0.42	0
0	-0.69	0	0	0	0	0	0	0
0.44	0.3	0	-0.58	-0.31	0	0	0	0

0.76	0	0.39	0	0	0	0	0	0
0.49	0	0	-0.56	-0.31	0	0	0	0
0.77	0	0.31	0	0	0	0	0	0
0	-0.95	0	0	0	0	0	0	0
0.55	0	0	-0.67	-0.37	0	0	0	0
0.91	0	0	0	0	-0.27	0	0	0
0.67	0	-0.25	-0.28	0.56	0	0	0	0
0.64	0	-0.59	0	-0.33	0	0	0	0
0.82	0	0.46	0	0	0	0	0	0
0	0	0	0	-0.21	-0.46	0	0.37	0

Table 8: The input matrix of loadings L for simulations by basic model

Next, to this structure, we add some randomness by simulation of 9 (number of factors) independent uniform random variables with mean equal to one and variance equal to $\frac{1}{3}$; every random variable is for each of 9 factors and we multiply each of 9 columns of matrix L by the corresponding random value. The choice of mean equal to one is explained by the wish in average get same loadings as in matrix L and variance is chosen equal to $\frac{1}{3}$ because we want to get in rare cases matrix L where sum of squared loadings exceeds $n\,(n = 42)$ (sum of $n \times 9$ squared loadings is equal to sum of 9 largest eigenvalues of correlation matrix and the sum is not greater than number of variables n).

If original data set A has missing values, we need some mechanism to bring absence of some values in data set S. If we use this model to simulate data sets, the model doesn't provide us with missing elements in data. So, again, we need to do something with simulated records that missing elements are presented in data. To add missing elements, we calculate relative frequency of missing elements for every variable of original data set A and apply these frequencies as estimates of probabilities that some variable has missing element. Then we take simulated record and use random number generator to generate n independent random values to determine if certain variable takes missing value or we leave the given value. But of course, absence in one variable can correlate with absence in another and the mechanism above doesn't take this into account.

The button called *Basic model* on the application form has to be pushed to simulate data set using this method. A matrix of loadings must be located in workspace of Matlab under name L. A number of factors on default is equal to a number of columns of matrix L. When a user pushes the button, the function *pushbutton callback* finds matrix L in Matlab's workspace; if it can't find it, error message appears. Then the function *SimulationFAM* is executed and the loop in the body of this program simulates m records that are distinct from records of matrix A. The loop on every iteration simulates n values using the factor model with normal distribution of factors and normal errors. Then n continuous values of simulated record are transformed to discrete values from the set of possible values by the function *TransformToDiscret*. The set of possible discrete values of all n variables are found by function *RetFreq* and every column

of output matrix is sorted in ascending order by function *SortOfVariab*. The principle behind the transformation is to find the closest allowed value to a given continuous value produced by factor model. The function *Check2* verifies if a simulated record is distinct from records of matrix A. The output of the function *SimulationFAM* is matrix S which is stored in workspace of Matlab under name S. It's all eigenvalues are calculated and plotted, error is found by function *PlotEigErr*.

The results I got for 9 factors is following: minimum value for MSE of factor loadings is 17.95, where 50 different matrices of L were simulated and with each matrix L, data set S was simulated 100 times. We have 378 factors loadings and average squared error in this case for one loading coefficient is $\frac{17.95}{378} = 0.05 = (0.22)^2$. Then I tried to use model with three factors instead of nine. The result for MSE is 4.37. Here, average square error for one loading coefficient is $\frac{4.37}{126} = 0.04 = (0.19)^2$. In addition, I used uniform distribution for factors instead of normal. The MSE in this case for three factors is larger than for model with normal factors and equal to 7.57.

The code behind the button and all embedded functions with comments is shown below.

```
% --- Executes on button press in pushbutton18.
function pushbutton18_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton18 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% starting initializations
A=handles.current_data;
n=handles.n;
m=handles.m;
lamda=handles.lamda;
dmissing=handles.dmissing;

% determine matrix of loadings
try
        L=evalin('base','L');
catch
        error;
end%{try}

k=size(L,2);
[S]=SimulationFAM(A,L,m,n,k);

% check if S has missing elements
if dmissing
        dmissing=false;
        for i=1:m
             for j=1:n
                      if isnan(S(i,j))
                           dmissing=true;
                      end%{if}
              end%{for j}
         end%{for i}
 end%{if dmissing}
```

```
assignin('base','S',S);
% plot eigenvalues and squared error
 [lam,error]=PlotEigErr(S,lamda,dmissing,n,'black','--bo');
% calculated error is shown on application form
set(handles.text16,'String',error);
msgbox('Output is matrix S');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [S]=SimulationFAM(A,LA,m,n,t);
%Input - A is m times n matrix containing data set (every row is
%                an record containing n different variables)
%        m is number of rows in A (number of records)
%        n is number of columns in A (number of variables in
%             every record)
%        t is number of factors
%        LA is n times t a matrix of factor loadings of data set
%                     A for t factors
%Output - S is m times m matrix containing simulated records; all
%             records are distinct from those records on which
%             bases they were simulated;

% we simulate matrix S using factor model for t factors;
% every factor is standard normal and independent of each other;
% error term is normal with zero mean and standard deviation
% is estimated by sample standard deviation; factor loadings
% are given in matrix LA

% calculate mean and standard deviation of every variable of
% data set A
mn=mean(A);
mn=mn';
stdev=std(A);

% determine possible values of each of n variables
% matrix V contains all possible distinct elements of each variable
[F,V]=RetFreq(A,m,n);
[M,indexM]=SortOfVariab(F,V,n);

% simulate m records by factor model
k=1;
while k<m+1
      c=true;
      while c
            % assign values to each of t factors as standard
            % normal
            for i=1:t
                  Factor(i,1)=randn;
            end%{for}

            % assign values to error terms of each variable
            for j=1:n
                  eps(j,1)=randn*stdev(1,j);
            end;

            % calculate simulated record by given factor model
```

```
            sim=mn+LA*Factor+eps;
            sim=sim';

            % k-th simulated record is written in matrix S
            [sim]=TransformToDiscrete(sim,M,indexM);
            % check if we have such record in matrix A
            [b]=Check2(A,sim,m,n);
            if b
                c=false;
                if dmissing
                    [sim]=AddMissEl(sim,F,m,n);
                end%{if}
            end%{if}

        end%{while c}
        S(k:k,1:n)=sim;
        k=k+1;
end%{while k}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [sss]=TransformToDiscrete(sim,M,indexM);
%Input - sim is simulated record
%            M is h times n matrix containing different unequal
%                  values of elements encountered in certain
%                  column of A (all elements of every column of M
%                  are sorted in ascending order);
%             indexM is one times n array that shows how many
%                       distinct values each variable can take
%Output - sim is simulated record where all elements were
%                       replaced by the closest elements of A

% we have simulated record; but elements are not from the same
% range as elements of A; so for every variable we find the closest
% value in matrix M and we replace element in sim by it

n=size(sim,2);
for i=1:n
    % if element is lower than the smallest element of M, we assign
    % the smallest of M to it; if element is greater than the
    % largest element of M we assign the largest of M to it
    if (sim(1,i)<M(1,i)) | (sim(1,i)==M(1,i))
        t=M(1,i);
    else
        if (sim(1,i)>M(indexM(1,i),i)) | (sim(1,i)==M(indexM(1,i),
            i))
            t=M(indexM(1,i),i);
        else
            k=1;
            % determine between what two elements of M our
            % element and we assign the nearest to t
            while sim(1,i)>M(k,i)
                k=k+1;
            end%{while}
            d1=M(k,i)-sim(1,i);
            d2=sim(1,i)-M(k-1,i);
            if d1>d2
```

```
                    t=M(k-1,i);
                else
                    t=M(k,i);
                end%{if}

            end%{if second}

        end%{if first}

        sss(1,i)=t;
end%{for}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [M, indexM]=SortOfVariab(F,V,n);
%Input  - V is h times n matrix containing different unequal
%              values of elements encountered in certain column
%              of A
%         n is number of columns in A (number of elements in every
%            record)
%         F is h times n matrix containing frequencies of every
%            distinct element in certain column of A; the concrete
%            value of  h is determined during execution of code
%Output - M is h times n matrix containing different distinct
%                    values of elements encountered in certain
%                    column of A (all elements of every column of V
%                    are sorted in ascending order);
%         indexM is one times n array that shows how many
%                         distinct values each variable can take

len=size(F,1);

% for every column of matrix V, we find how many distinct
% values each variable can take and we sort all possible
% values of each variable in ascending order and place
% the result in matrix M
for i=1:n
    k=1;
    c=true;
    while c
        k=k+1;
        if (k==len) | (F(k,i)==0)
            c=false;
        end%{if}
    end%{while}

    if (k<len)
        k=k-1;
    end%{if}

    indexM(1,i)=k-1;
    B=V(2:k,i:i);
    B=sort(B);
    M(1:k-1,i:i)=B;
end%{for}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
function [sim]=AddMissEl(sim,F,m,n);
% add missing elements in simulated record;
% probability for every variable is estimated by
% relative frequency of missing elements;
for i=1:n
    if F(1,i)>0
        p=F(1,i)/m;
        x=rand;
        if x<p
            sim(1,i)=NaN;
        end%{if}
    end%{if}
end%{for}
```

5 The fourth method: multiple imputation by chained equations

The fourth method is based on one of the imputations methods. The simulation process is the following: we simulate initial record by independently modeling one element from each column of data set A, where probability that some value occurs is estimated by relative frequency of that element in that column. These n elements compose an initial record and we add it to original data set A. Then we imagine like we don't have the first element of new record and we use one of imputation techniques to fill in element. Multiple imputation by chained equations (MICE) (Van Buuren) is used to impute an element where stochastic regression is employed as basic tool. MICE works in the following way: we go to the first element of new record and given the rest $n-1$ elements of this record, we impute an element instead of current element. To find the value for replacement, we employ stochastic regression. We regress the first column on the remaining columns and find coefficients of regression. Using this coefficients, we find predictions for each of $m+1$ records given $n-1$ elements. Then, we add to every prediction noise with zero mean and standard deviation equal to residual standard deviation from regression. The regression gives us $m+1$ predictions and all these predictions take some real values, but elements from data set A belong to certain range and we can't replace values in record by predictions. So we employ predictive mean matching principle to get values from the same range as data set A. For some odd positive number k, we find k records with closest predictions to our record's prediction and choose median value between them. And we replace our value by value from corresponding column of median record. Then we go to the second column and repeat same procedure. When we finish such procedure with the last column, we have made one time imputation. And we repeat the same procedure from the beginning to the end multiple times. If original data set A has missing values, we can impute elements using same method where we can use column means as initial values for missing elements.

During my simulations I used linear regression and coefficients were esti-

mated by ordinary least squares. But to do proper regression estimation, that is, avoid multicollinearity, I checked if all columns are linear independent in our data set A. It turns out that three columns of data set A are in some linear combinations of the rest 39. So I excluded linear dependent columns and I did regression only with independent columns. But after I got simulated record, I added linear dependent elements to it to get record with full number of columns.

At the beginning of simulations I used instead of stochastic regression regression without noise. But if I chose all 38 columns as regressors (39 from 42 are independent columns), I didn't get any variability. For example, in the first column I got almost always value 102 or 114. (If I used 35 regressors, then elements were more variable). So I tried to use stochastic regression to get variability and, in addition, stochastic regression gives unbiased parameter estimates under missing at random mechanism (Enders, p.46).

The button called *MICE* on the application form performs simulations by MICE method. In the pop-up menu above the button, a user has to choose desirable odd number of how many neighbors to select. When a user pushes the button, two dialog boxes appear consequently: one to input number of imputations, another to input number of simulated records. All inputs are checked on correctness by function *InputNum*. Then the given value of number of neighbors is taken and the function *SimulationMICE* is called. First, we check if data set A has missing elements, if we have them, we replace them by rational number which is equal to maximum of all values of data set A plus one. So instead of missing elements, we have some concrete value. We do that because if we don't have all numbers, we can't do proper regression and, in addition, we want that missing elements are presented in simulated records. When we do imputations, it is possible on some step to impute this concrete value and that means we have missing element there. Second, the function determines all linear independent columns of A by function *LinIndep* and then loop in the program's body simulates records given number of times. The loop on every iteration calls the function *MTiCE* which does imputations given number of times by MICE method and it calculates k closest predictions and finds median distance. The function does all replacements as well. The body of the program *MItCE* contains two functions: one is called *PRegression*, this function performs linear regression and calculates predictions; the second is called *Check2*, this function is to check if a simulated record is distinct from records of data set A. When we have simulated a distinct record, we add linear dependent columns to it by the function *AddLinDepend* and, in addition, if one of the record's value is equal to maximum value of all elements of A plus one, then we replace it by missing value NaN. The output of the function *SimulationMICE* is matrix S which is stored in workspace of Matlab. All eigenvalues are calculated and plotted, error is shown on application form by the function *PlotEigErr*.

The simulation lasts long time. To simulate 100 records with $k = 5$ and 100 imputations takes 4500 seconds. I simulated three sets under different inputs and the following tables shows 9 largest eigenvalues and square error of eigenvalues and factor loadings for nine factors in each of three cases.

2.69	2.34	1.99	1.93	1.77	1.69	1.63	1.58	1.50

Table 9: The 9 largest eigenvalues of data set (100 records) simulated by MICE when k=5 and the number of imputations is 100

SE_λ	SE_l
111.1	40.30

Table 10: The SE of eigenvalues and factor loadings for 9 factors of data set (100 records) simulated by MICE when k=5 and the number of imputations is 100

2.40	2.38	2.09	1.81	1.74	1.64	1.57	1.57	1.52

Table 11: The 9 largest eigenvalues of data set (200 records) simulated by MICE when k=1 and the number of imputations is 10

SE_λ	SE_l
117.12	41.6

Table 12: The SE of eigenvalues and factor loadings for 9 factors of data set (200 records) simulated by MICE when k=1 and the number of imputations is 10

3.08	2.66	2.35	2.26	2.03	1.87	1.80	1.73	1.56

Table 13: The 9 largest eigenvalues of data set (100 records) simulated by MICE when k=1 and the number of imputations is 100

SE_λ	SE_l
101.78	47.9

Table 14: The SE of eigenvalues and factor loadings for 9 factors of data set (100 records) simulated by MICE under k=1 and number of imputations is 100

The result tells that this approach doesn't lead us to data sets with desirable properties. The simulated data set S doesn't preserve dependence between variables of data set A and SE is very large. The result, we got, might depend on the rule we set initial record. We selected elements for initial record by random independent draws. Maybe different rule to choose initial record provides us with better result, so I tried to set initial record by the following method: first, we choose randomly a column of matrix X (all independent columns of A) and then we choose an element from the selected column; it is our first element of initial record and we place it in chosen column. After we randomly choose the next column and regress the selected column on previously chosen column;

we find coefficients of regression and calculate prediction by the regression and convert the prediction to the closest value from the set of possible values for given column in X; it is the second element of initial record and we place it in chosen column. We continue to do the same procedure: we regress randomly chosen column on previously chosen columns until we have selected all columns of X and determined elements in all columns of initial record. Then I did simulations by MICE method where initial record is selected by just described method. The following tables show value of eigenvalues and square errors of eigenvalues and factor loadings for nine factors. The results are certainly better than before but not enough: square error of loadings for nine factors is still large.

14.44	3.09	2.41	2.17	1.50	1.42	1.27	1.22	1.17

Table 15: The 9 largest eigenvalues of data set (200 records) simulated by MICE when k=1 and the number of imputations is 5

SE_λ	SE_l
3.59	32.11

Table 16: The SE of eigenvalues and factor loadings for 9 factors of data set (200 records) simulated by MICE when k=1 and the number of imputations is 5

14.35	3.38	2.56	2.13	1.70	1.45	1.36	1.34	1.29

Table 17: The 9 largest eigenvalues of data set (100 records) simulated by MICE when k=1 and the number of imputations is 100

SE_λ	SE_l
3.05	28.82

Table 18: The SE of eigenvalues and factor loadings for 9 factors data set (100 records) simulated by MICE when k=1 and the number of imputations is 100

15.91	3.90	2.81	2.53	1.94	1.58	1.51	1.36	1.28

Table 19: The 9 largest eigenvalues of data set (100 records) simulated by MICE when k=1 and the number of imputations is 200

SE_λ	SE_l
10.99	36.18

Table 20: The SE of eigenvalues and factor loadings for 9 factors of data set (100 records) simulated by MICE when k=1 and the number of imputations is 200

15.53	3.16	2.70	2.19	2.11	1.90	1.62	1.59	1.42

Table 21: The 9 largest eigenvalues of data set (100 records) simulated by MICE when k=5 and the number of imputations is 100

SE_λ	SE_l
9.18	28.04

Table 22: The SE of eigenvalues and factor loadings for 9 factors of data set (100 records) simulated by MICE under k=5 and number of imputations is 100

It is clear that results we got depend on choice of rule that selects initial record. The second rule where elements in initial record were made dependent by regressing is better than the first rule where all elements were independent. The code behind the button and all embedded functions with comments is shown below.

```
% --- Executes on button press in pushbutton17.
function pushbutton17_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton17 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% input the number of imputations
str='Enter the number of multiple imputations';
type='integer';
answer1=InputNum(str,0,100000,type);
if ~isempty(answer1) & (answer1~=-1)
        % input of how many records to simulate
        str='Enter the desirable number of simulated records';
        type='integer';
        answer2=InputNum(str,0,1000000,type);
        if ~isempty(answer2)& (answer2~=-1)
              % starting initialization
               k=answer1;
               numberSim=answer2;
               A=handles.current_data;
               n=handles.n;
               m=handles.m;
               dmissing=handles.dmissing;
               lamda=handles.lamda;
               % a user has to choose in pop-up menu above the
               % button what value to use for predictive matching
```

```
                % principle
                popupVal=get(handles.popupmenu4,'Value');
                switch popupVal
                        case 1%case 3
                            meanP=3;
                        case 2 %case 1
                            meanP=1;
                        case 3 %case 5
                            meanP=5;
                        case 4 %case 7
                            meanP=7;
                 end%{switch}
                [S]=SimulationMICE(A,m,n,k,meanP,numberSim,dmissing);

                % check if S has missing elements
                 if dmissing
                    dmissing=false;
                     for i=1:m
                        for j=1:n
                                if isnan(S(i,j))
                                    dmissing=true;
                                end%{if}
                            end%{for j}
                        end%{for i}
                 end%{if dmissing}

                 assignin('base','S',S);
                 % plot eigenvalues and squared error
                 [error,error]=PlotEigErr(S,lamda,dmissing,n,'black',
                     '--bo');
                 % calculated error is shown on application form
                 set(handles.text16,'String',error);
                 msgbox('Output is matrix S');
        end%{if ~isempty(answer2)}
end%{if ~isempty(answer1)}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [S]=SimulationMICE(A,m,n,num,meanP,numberSim,dmissing);
%Input - A is n times m matrix containing data set A
%        m is the number of rows in A
%        n is the number of columns in A
%        mum is the number of imputations
%        meanP is odd number thet determines number of closest
%                    neighbors
%        numberSim is a number of simulated records
%        dmisiing is boolean variable that shows if we have
%                        missing values in data set A
%Output - S is numberSim times m matrix containing data set of
%                    simulated records

% we simulate matrix S using multiple imputation by chained
% equations method; we add new record to matrix A; we simulate
% iid draws of elements from columns of A to fill initial values
% for the new record; after we imagine that we have missing
% element in the first variable of new record and we want to
% impute a value for the missing value; we use stochastic
```

```
% regression to impute a value; all remaining linear independent
% columns are used to estimate coefficients of linear regression
% and using this regression, we assign definite values to
% initial record and to all records of A; using these values, we
% choose odd number; (meanP) of the closest neighbors to
% the new record and find record with median value (ceil(odd/2));
% then we replace a value of first variable of new record by the
% corresponding value of found median record and we go to the
% next variable and so on.

% first we check if we have missing values; if we find some missing
% values, we replace them by max+1 where max is maximum value
% of all elements of matrix A
if dmissing
    v=max(A);
        maxV=v(1,1);
        for j=2:n
        if maxV<v(1,j)
            maxV=v(1,j);
        end%{if}
        end%{for}
        maxV=maxV+1;

        % we don't want to change something in matrix A, so we
        % create copy of A to replace missing values by max+1
        C=A;
        for i=1:m
          for j=1:n
            if isnan(C(i,j))
                C(i,j)=maxV;
            end%{if}
          end%{for j}
        end%{for i}
else
    C=A;
end%{if dmissing}

% some columns of matrix A can be linear dependent; in this
% case, we can't do proper regression (multicollinearity), so
% to prevent this, we extract linear independent columns from A;
% indIn consists of indexes of linear independent columns;
% indOut consists of indexes of linear dependent columns;
% alfa consists of coefficients that express dependent columns
% through independent;
% X is matrix of linear independent columns of A;
[X,indIn,indOut,alfa]=LinIndep(C,m,n);

% i shows the number of simulated records
i=1;
numberSim=numberSim+1;
while (i<numberSim)
     % we create new record by multiple imputation by
     % chained equations method
     [newrecord]=MItCE(X,m,num,meanP);
     % we restore initial number of columns by adding linear
     % dependent columns
     [newrecordFin]=AddLinDepend(newrecord,X,indIn,indOut,alfa,n);
```

```
      % if we get value equal to (max+1) in simulated matrix S,
      % that means we have there missing value, so we replace
      % max+1 by NaN
           if dmissing
            for j=1:n
                if newrecordFin(1,j)==maxV
                     newrecordFin(1,j)=NaN;
                end%{if}
            end%{for j}
      end%{if dmissing}

      S(i:i,1:n)=newrecordFin;
      i=i+1;
end%{while i}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [newrecord]=MItCE(X,m,num,meanP);
%Input - X is matrix that contains all linear independent columns
%                    of A
%          m is the number of rows in X
%          mum is the number of imputations
%          meanP is odd number that determines number of the closest
%                           neighbors
%Output - newrecord is simulated record through MICE method

% we simulate matrix S using multiple imputation by chained
% equations method;

% we create new record; initially, we assign n independent random
% elements from each column of A to the new record
r=rank(X);
ranEl=RanElem(X,m,r);

% create matrix Z by union of new record and matrix X;
% new record is placed on the last row of matrix Z
Z(1:m,1:r)=X(1:m,1:r);
Z(m+1:m+1,1:r)=ranEl;

% we impute values to the last row of matrix Z num times (we
% consider them like they are missing); one time imputation of
% the record starts from the first variable and ends at the last
% variable; to impute one element in certain column, we regress
% this column on others columns of matrix X; we use found
% regression coefficients to predict values of this column;
% using m+1 calculated predictions, we choose meanP closest
% neighbors to the last record and choose record with median
% value. we take a value from the given column in chosen
% median record and replace the value in that column of the
% last record by the value in the given column of chosen
% median record

k=0;% number of imputations
i=0; % number of variables

b=true;
```

```
% b checks if we have identical record in matrix A as
% simulated record
while (b)
      while (k<num)
             % i shows current column variable
             i=i+1;
             if i>r
                        k=k+1;
                        i=1;
             end%{if}
             % we find prediction using stochastic regression;
                % we regress i column on the rest columns
                [yfit]=PRegression(Z,m,r,i);

                % we use predictive mean matching principle to
                % replace element in column i; use regression
                % predictions to find meanP closest neighbors;
                % afterwards, we choose record that have median
                % value between meanP neighbors and we replace
                % value in column i of the last record by
                % corresponding value of found record
                 for j=1:m
                        distance(j,1)=abs(yfit(j,1)-yfit(m+1,1));
                 end%{for j}

                 % find meanP indexes of closest neighbors and then
                 % find median predictor's index;
                  t=0;
                  while (t < meanP)
                          min=distance(1,1);
                          ind=1;
                          for j=2:m
                                  if min>distance(j,1)
                                          ind=j;
                                          min=distance(j,1);
                                  end%{if}
                          end%{for j}
                          distance(ind,1)=10^6;
                          % contains minimums indexes
                           indM(1,t+1)=ind;
                           t=t+1;
                  end%{while t<}
                  % find median between closest neighbors
                   c=ceil(meanP/2);
                   w=indM(1,c);
                   Z(m+1,i)=Z(w,i);
           end%{while num}

           S=X(1:m,1:r);
           B=Z(m+1:m+1,1:r);
           % check if we have already such record in X
           [c]=Check2(S,B,m,r);

          % c is true if simulated record is distinct from
          % records data set A
           if c
                  b=false;
```

```
                newrecord=Z(m+1:m+1,1:r);
            end%{if c}
end%{b}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [X,indIn,indOut,alfa]=LinIndep(A,m,n);
%Input - A is m times n matrix containing data set
%            m is the number of rows in A
%            n is the number of columns in A
%Output - indOut includes indexed of dependent columns
%         indIn includes indexed of independent columns
%         X contains all linear independent columns of A
%         alfa contains coefficients that express linear
%              dependent columns through linear independent ones

% we find linear independent columns of A;
% in case we have linear independent columns, we can't
% do proper regression (multicollinearity), so to prevent this,
% we extract linear independent columns from A
X=A(1:m, 1:1);
indIn(1)=1;
j=2;
h=1;
for i=2:n
    r=rank(X);
    Y=A(1:m, i:i);
    Z=[X Y];
    if rank(Z)>r
        X=Z;
        indIn(j)=i;
        j=j+1;
    else
        indOut(h)=i;
        h=h+1;
    end%{if}
end%{for i}

% we find coefficients for expressing linear dependent
% columns through linear independent ones
r=rank(X);
if exist('indOut')~=0
    k=size(indOut,2);
    for i=1:k
        s=indOut(i);
        D=A(1:m, s:s);
        alfa(i:i,1:r)=(X\D)';
    end%{for i}
else
    indOut(1)=NaN;
    alfa(1:1,1:r)=NaN;
end%{if}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [newrecordFin]=AddLinDepend(newrecord,X,indIn,indOut,alfa,
    n)
```

```
%Input -  newrecord is simulated record without dependent columns
%         n is the number of columns in A
%         indOut includes indexed of dependent columns
%         indIn includes indexed of independent columns
%         X matrix that contains all linear independent columns
%                       of A
%         alfa is matrix that contains coefficients that explain
%                linear dependent from linear independent ones
%Output - newrecordFin is simulated record with full number
%                                    of columns

% we find values of linear dependent columns of new simulated
% record; we add linear dependent values using found before
% coefficients for expressing dependence that we stored in alfa
if ~isnan(indOut(1))
        % original number of columns
        for j=1:n
                newrecordFin(1,j)=0;
        end%{end}

        % first we place linear independent columns on it's
            original
           % place
        k=size(indIn,2);
        j=0;
        while (j<k)
                    j=j+1;
                    t=indIn(j);
                    newrecordFin(1,t)=newrecord(1,j);
        end%{while}

          % we place linear dependent columns on it's original
              place
          vec=alfa*(newrecord');
        k=size(indOut,2);
        j=0;
        while (j<k)
                    j=j+1;
                    t=indOut(j);
                    newrecordFin(1,t)=vec(j,1);
        end%{while j}
else
          newrecordFin=newrecord;
end%{if}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [yfit]=PRegression(Z,m,r,i);
%Input -  m is the number of records in original data set A
%         i is index of column which is dependent variable in
%             regression
%         Z matrix that contains all linear independent columns
%                   of A plus one new record
%         r is rank of Z
%Output - yfit ais vector of predictions by stochastic regression

% we regress i column of Z on the rest columns of Z using linear
```

```
% regression
X=Z;

% construct dependent variable
Y=Z(1:m+1,i:i);

% construct independent variable
G=X(1:m+1,1:i-1);
X(1:m+1,2:i)=G;
X(1:m+1,1:1)=1;

% linear regression
beta=inv(X'*X)*(X'*Y);
yfit = X*beta;

% estimate standard deviation
res=Y-yfit;
s=std(res);

% add normal noise to every prediction
for i=1:m
    e=randn*s;
    yfit(i,1)=yfit(i,1)+e;
end%{for i}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [rel]=RanElem(X,m,r);
% choose randomly elements from each
% column of A (independent draws)
for i=1:r
    x=rand(1);
    indc=ceil(x*m);
    rel(1,i)=X(indc,i);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [SSS]=RanElem2(X,m,r);
% we find initial record;
% first, we choose randomly an column of matrix X and
% then we choose an element from the selected column;
% after we randomly choose the next column and regress
% the selected column on previous chosen column;
% we find coefficients of regression and calculate
% prediction by the regression and transfer the prediction
% to the closest value from the set of possible values for
% given column  in X. we continue to do the same procedure:
% where we regress randomly chosen column on previous chosen
% columns until we have selected all columns of X and calculated
% all elements of array SSS[1:1,1:r];

% determine all distinct values of every variable of X
[F,V]=RetFreq(X,m,r);
[M,indexM]=SortOfVariab(F,V,r);

% starting initializations
```

```
% array CONDB shows indexes of chosen columns
for i=1:r
    CONDB(1,i)=false;
    SSS(1,i)=0;
end%{for i}

% fill each variable of array SSS(1:1,1:r)
p=2;
i=r;
while (i>0)
      % randomly select one of not already chosen columns of X
      [t,CONDB]=DiscreteRand1(i,CONDB,r);
      % at the beginning, we find  select by randomness the first
      % element; then we use linear regression to determine
      % next elements
      if i==r
          x=rand(1);
          indc=ceil(x*m);
        % initial record
          SSS(1,t)=X(indc,t);
          % to find correctly predictions, we need array with
          % certain order which is  multiplied by corresponding
          % coefficients of regressions
          ggg(1,1)=X(indc,t);
          % matrix of regressors
          Regr=X(1:m,t:t);
      else
          % dependent variable of regression
          Y=X(1:m,t:t);
          % we use linear regression and find coefficients by OLS
          beta=inv(Regr'*Regr)*(Regr'*Y);
          % find predictions
          yfit = ggg*beta;
          % transform prediction to the nearest value from the set
          % of possible values of X
          if (yfit<M(1,t)) | (yfit==M(1,t))
                x=M(1,t);
        else
                if (yfit>M(indexM(1,t),t)) | (yfit==M(indexM(1,t),
                    t))
                      t=M(indexM(1,t),t);
                else
                      k=1;
                      % determine between what two elements of M
                      % our element and we assign the nearest to
                            t
                      while yfit>M(k,t)
                            k=k+1;
                      end%{while}
                      d1=M(k,t)-yfit;
                      d2=yfit-M(k-1,t);
                      if d1>d2
                          x=M(k-1,t);
                      else
                          x=M(k,t);
                      end%{if}
                end%{if second}
```

```
            end%{if first}
            % add new column to regressors matrix
            Regr=[Regr,Y];
            % we write found element in initial record
            SSS(1,t)=x;
            % we write found element in array ggg
             ggg(1,p)=x;
             p=p+1;
        end%{if}
        i=i-1;
end%{while i}
```

6 Conclusion

As it turned out, the best method is method based on k-neighborhood principle. The rest three methods provide unsatisfactory results. The data sets simulated by those three methods can't preserve structure embedded in data set A. Those methods contain different assumptions that likely to be wrong. In the second method, it might be that we can't estimate properly conditional probabilities by relative frequencies because of small number of records of data set A. In the third method, we might have made wrong assumptions about distributions of factors or true number of factors or true factor loadings matrix. In the last method, we used linear regression, it might be some different form should have been used instead of linear one. On the contrary, k-neighborhood method is simple to implement, running time is not large and provides with very good result. The method considers some initial record and it's k neighbors and they are similar to each other and during simulations we get some distinct record but it can't be different too much from those on which base it is simulated. So dependence between variables of data set A is preserved in simulated data set. In addition, when we did simulations by k-neighborhood method, we used only values that n variables of data set A can take. In the last two methods, we used data transformation from continuous values to values from set of possible values by finding nearest allowed value. It might be that during transformations, we loose some structure and these transformations should be done in more sophisticated way.

7 Appendix

I added to the application the button called *"Factor Analysis"* to perform factor analysis. The data set A has negative eigenvalues and in this case the implemented in Matlab function doesn't work. The button performs principal component factor analysis on data set chosen in the pop-up menu which is located above the button. Using the check-boxes a user can provide it's preferable number of factors or can use rule that selects the number of factors equal to the number of eigenvalues that have value greater than one. The button *Update*

determines current data in Matlab's workspace and shows all data in the list box. The output of factor analysis is stored in workspace as matrix *OUTPUT-load*. The last row of a matrix shows eigenvalues; the column before the last shows communalities h_i^2 and the last column shows specific variances ψ_i^2. If data set has missing values, covariance matrix is found by maximum likelihood estimation via expectation-maximization algorithm. The button *Rotate* makes rotation of factor loadings of matrix L by the one of two methods (varimax or promax) that can be selected in check-boxes below the button.

The function *MainProgram3* was written to perform calculations of MSE of eigenvalues and factor loadings. The user has to choose in code desirable method of simulation. Simulations occurs *num* times; then mean and variance are estimated and we calculate MSE where the values of eigenvalues and factor loadings of data set A are taken as a true values. The function *CheckSignEV* changes eigenvectors v by $(-1) * v$ when we calculate factor loadings, if it decreases MSE for corresponding factor loadings.

The button *FileOutput* creates text file and prints in it elements of matrix S, which is stored in the workspace of Matlab. The code behind all buttons and embedded functions is attached to master thesis.

The part of code for application which was not appeared above is shown below.

```
% --- Executes on button press in pushbutton12.
function pushbutton12_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton12 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
update_listbox(handles);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function update_listbox(handles)
var=evalin('base','who');
set(handles.listbox2,'String', var);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% --- Executes on button press in checkbox3.
function checkbox3_Callback(hObject, eventdata, handles)
% hObject    handle to checkbox3 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% Hint: get(hObject,'Value') returns toggle state of checkbox3
set(handles.checkbox5,'Value',0);

% --- Executes on button press in checkbox5.
function checkbox5_Callback(hObject, eventdata, handles)
set(handles.checkbox3,'Value',0);

% --- Executes on button press in checkbox1.
function checkbox1_Callback(hObject, eventdata, handles)
set(handles.checkbox2,'Value',0);
```

```
% ---- Executes on button press in checkbox2.
function checkbox2_Callback(hObject, eventdata, handles)
set(handles.checkbox1,'Value',0);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [answer]=InputNum(str,a,b,type);
%Input - str is a text that appears on dialog box
%        [a,b] is a interval for a number
%        type takes two values integer or real
%Output - answer is a returned user's input checked on correctness

% the function creates a dialog box and user has to input a number
% from a range [a,b]. a user imposes condition on number's type
% through parameter type .
prompt = {str};
dlg_title = 'Input for a number';
num_lines= 1;
def     = {'1'};
answer  = inputdlg(prompt,dlg_title,num_lines,def);

if ~isempty(answer)
% convert from string type to number type
        answer=str2num(answer{:});
        % check if input is a positive integer or real number
            between
        % a and b
        switch type
             case 'integer'
                 if  (answer < a) | (answer > b) |(rem(answer,1)
                     ~=0)
                      msgbox('input must be an integer number in
                          certain range');
                      answer=-1;
                 end%{if}
             case 'real'
                 if (answer < a) | (answer > b)
                      msgbox('input must be an integer number in
                          certain range');
                      answer=-1;
                 end%{if}
        end%{switch}
end%{if}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [lam,error]=PlotEigErr(S,lamda,dmissing,n,color1,color2);
%Input - S is matrix with data set;
%           lamda is array of eigenvalues. to find error, we
    compare
%                       these eigenvalues to eigenvalues of
    correlation
%                       matrix of data set S;
%           dmissing is boolean variable that shows if we have
%                       missing data in S;
%            n is number of columns of matrix S;
%Output - lam is array of all eigenvalues of S;
```

```
%               error is a sum of squared errors of corresponding
%                       eigenvalues of S and given in input
%                       vector lamda;

% function finds eigenvalues of correlation matrix of data set S
% and  plots them. in addition, it finds sum of squared errors
% of corresponding eigenvalues of S and vector lamda given
% in input.

% check row size of S (row size must be greater than n to find
% n eigenvalues)
k=size(S,1);
if k<n
    msgbox('row dimension of S is too small to find n eigenvalues')
        ;
else
        % if we have missing data, maximum likelihood method is
        % used to find covariance matrix (expectation
        % maximization algorithm)
        if dmissing
                [SMean, SCovariance] = ecmnmle(S);
                [ExpSigma, SCorr] = cov2corr(SCovariance);
                lam=eig(SCorr);
        else
               G=corrcoef(S);
                lam=eig(G);
        end%{if}

        % plot eigenvalues of matrix S
        plot(lam,color2,'LineWidth',1,...
                'MarkerEdgeColor',color1,...
                'MarkerFaceColor',color1,...
                'MarkerSize',4)
        xlabel('eigenvalues');

        % calculate sum of squared errors of corresponding
        % eigenvalues of S and lamda
        error=0;
        for i=1:n
            error=error+(lamda(i)-lam(i))^2;
        end%{for}
end%{if}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% --- Executes on button press in pushbutton11.
function pushbutton11_Callback(hObject, eventdata, handles)

% principal component factor analysis

% choose data set selected in pop-up menu by user;
entry=get(handles.popupmenu3,'String');
index=get(handles.popupmenu3,'Value');

% find selected in pop-up menu data set in workspace
varName=char(entry(index));
try
```

```
    varValue=evalin('base',varName);
catch        exception
error(['cc ''' varName '''jkjjj']);
end%{try}

% data set is in matrix A
A=varValue;
m=size(A,1);
n=size(A,2);

fmissing=false;
i=0;
j=0;
while i<handles.m
    while j<handles.n
        if isnan(A(i+1,j+1))
            fmissing=true;
        end%{if}
        j=j+1;
    end%{while}
    i=i+1;
end%{while}

% check if we have missing data in A; in this case, we use
% maximum likelihood method to estimate correlation
% matrix of data set A
if fmissing
     [AMean, ACovariance] = ecmnmle(A);
     [ExpSigma, G] = cov2corr(ACovariance);
else
     G=corrcoef(A);
end%{if}

% to perform factor analysis, it is necessary to choose number of
% factors. a user has two options: input his own preferable
% number or choose rule of thumb (the number of eigenvalues
% that have value greater than one)
value=get(handles.checkbox1,'Value');
% the first option: input preferable number
if value==0
            % input number of factors
            str='Enter number of factors:';
            type='integer';
            answer=InputNum(str,0,n,type);
            if ~isempty(answer) & (answer~=-1)
                    k=answer;
                    [OUTPUTload,Load]=ChooseEigLoad(G,k,n);
                    assignin('base','OUTPUTload',OUTPUTload);
                    msgbox('Output is matrix OUTPUTload');
            end%{if ~isempty(answer)}
else
            % number of eigenvalues > 1 rule
            [lam]=eig(G);
            k=0;
            for j=1:n
                if lam(j,1)>1
                    k=k+1;
```

```
            end%{if}
        end%{for}

        [OUTPUTload,Load]=ChooseEigLoad(G,k,n);
        assignin('base','OUTPUTload',OUTPUTload);
        msgbox('Output is matrix OUTPUTload');
end%{if else}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [OUTPUTload,Load]=ChooseEigLoad(G,k,n);
%Input - G is n times n correlation matrix
%           k  is a number of factors
%           n is number of columns in A (number of variables in
%               every record)
%Output - OUTPUTload is output of factor analysis

% V matrix of eigenvectors and lam is matrix where diagonal
% elements are eigenvalues
[V,lam]=eig(G);

% find k largest eigenvalues and their corresponding eigenvectors
i=1;
while i<k+1
      maxL(1)=lam(1,1);
      maxL(2)=1;

      for j=2:n
            if lam(j,j)>maxL(1)
                  maxL(1)=lam(j,j);
                  maxL(2)=j;
            end%{if}
      end%{for}

      EIGvect(1:n,i:i)=V(1:n,maxL(2):maxL(2));
      EIGval(1,i)=maxL(1);
      i=i+1;
      lam(maxL(2),maxL(2))=-1;
end%{while}

% find factor loading for k factors
for j=1:k
    Load(1:n,j:j)=(EIGval(1,j))^(1/2)*EIGvect(1:n,j:j);
end%{if}

% find communalities and specific variances
H=Load*Load';
KSI=G-H;
ERR=G-(H+eye(n)*KSI);

HH=diag(H);
KSII=diag(KSI);


OUTPUTload(1:n,1:k)=Load;
for j=1:k
    OUTPUTload(n+1:n+1,j:j)=(EIGval(1,j));
```

```
end%{for}

for j=1:n
    OUTPUTload(j:j,k+1:k+1)=HH(j,1);
end%{for}

for j=1:n
    OUTPUTload(j:j,k+2:k+2)=KSI(j,1);
end%{for}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% --- Executes on button press in pushbutton16.
function pushbutton16_Callback(hObject, eventdata, handles)
% factor rotation by selected method of factor loadings in L matrix
try
  L=evalin('base','L');
catch
  error;
end
% determine which method is selected
value=get(handles.checkbox3,'Value');
if value==0
   RL=rotatefactors(L,'method','varimax');
else
   RL=rotatefactors(L,'method','promax');
end%{if}
% output of rotation is matrix RL
assignin('base','RL',RL);
msgbox('Output is matrix RL');
```

References

[1] W. Haerdle, L. Simar, *Applied Multivariate Statistical Analysis*. Springer-Verlag Berlin Heidelberg, 2nd Edition, 2007.

[2] S. Ross, *Simulation*. Amsterdam:Elsevier Acad. Press, 4th Edition, 2006.

[3] C. Enders, *Applied missing data analysis*. Guilford press, 2010.

[4] R. Little, D. Rubin, *Statistical analysis with missing data*. JOHN WILEY SOHNS, 1987.

[5] Matlab, *help, www.mathworks.com/help*.

[6] J.Mathews, K.Fink, *Numerical methods using Matlab*, Prentice Hall, 3rd Edition, 1999.

[7] Latex, *stackoverflow.com/questions*, *tex.stackexchange.com/questions*.

[8] S. Van Buuren, C. Oudshoorn *Flexible multivariate imputation by MICE*, Leiden: TNO Prevention and Health, report PG/VGZ/99.054, 1999.

Declaration of Authorship

I hereby certify that the thesis I am submitting is entirely my own original work except where otherwise indicated. I am aware of the University's regulations concerning plagiarism, including those regulations concerning disciplinary actions that may result from plagiarism. Any use of the works of any other author, in any form, is properly acknowledged at their point of use.

Student's signature:

Name (in capitals):

Date of submission: